Networking

Prof. Dr.-Ing. Holger Hermanns

Dependable Systems & Software
Saarland University

Summer 04

Session B: Reliable Data Transfer

Reliable Data Transfer

Read chapter 4 of [Holzmann 91] & chapter 3.3 of [Tanenbaum]

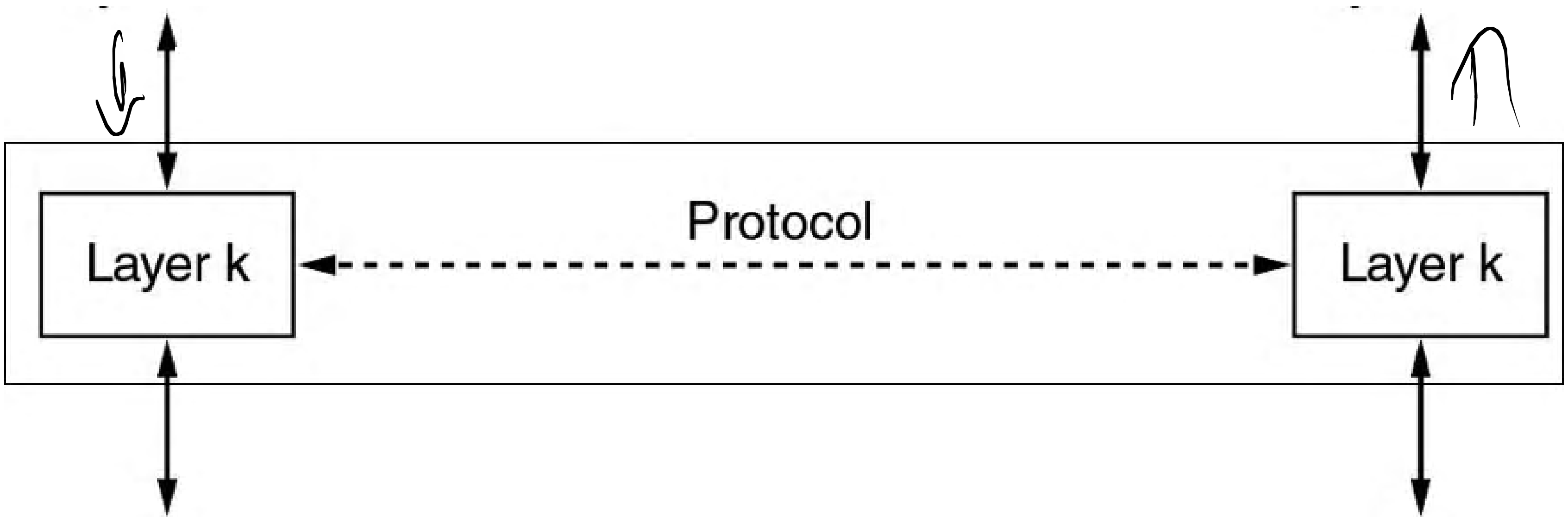
Our goal today:

- develop a full reliable data transfer protocol
- approach:
  - add complexity step-by-step
  - introduce the ingredients

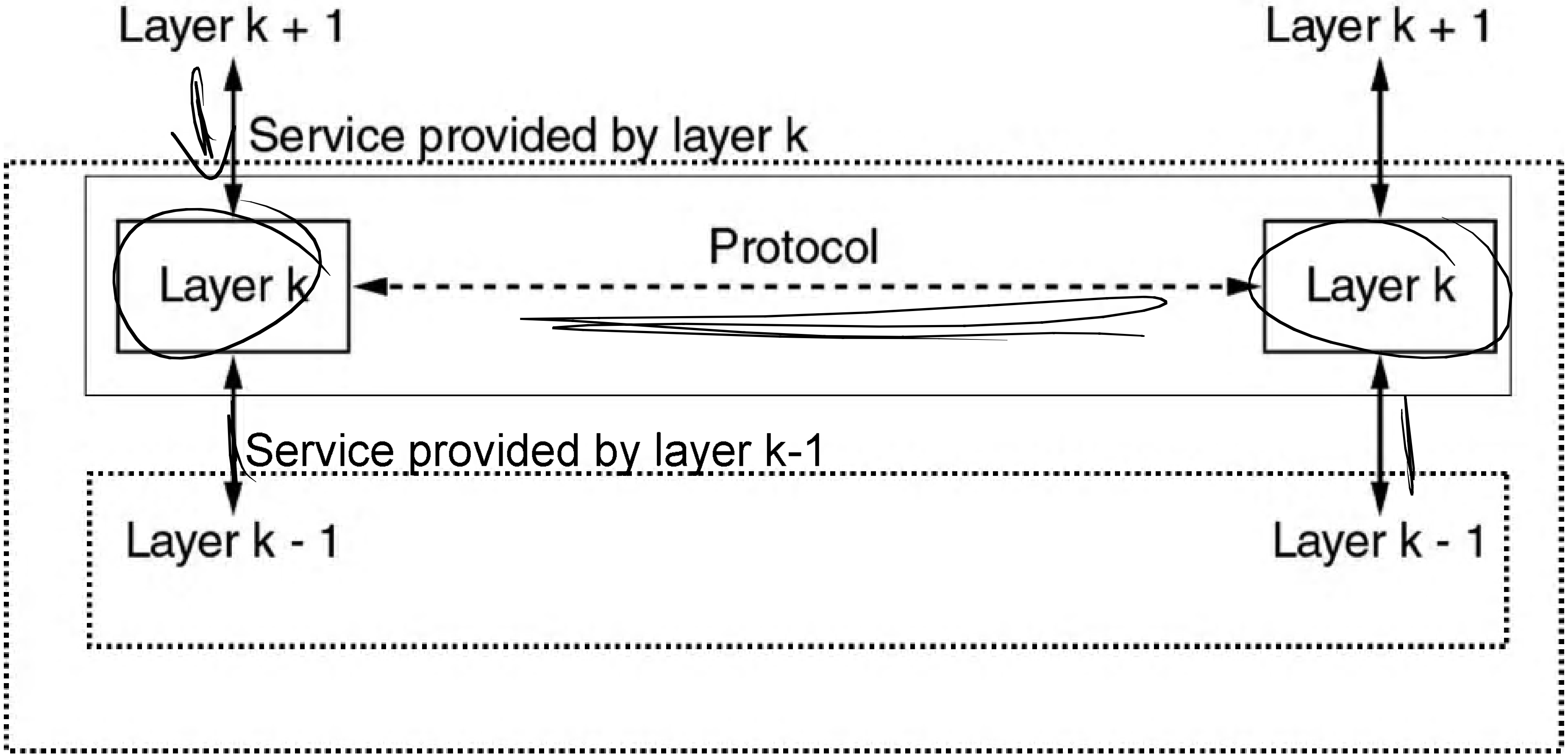
Overview:

- Unrestricted simplex protocol
- simplex stop-and-wait
- windows, sequence numbers
- unreliable channels
- timers
- alternating-bit

Recall: Protocol, Environment and Services

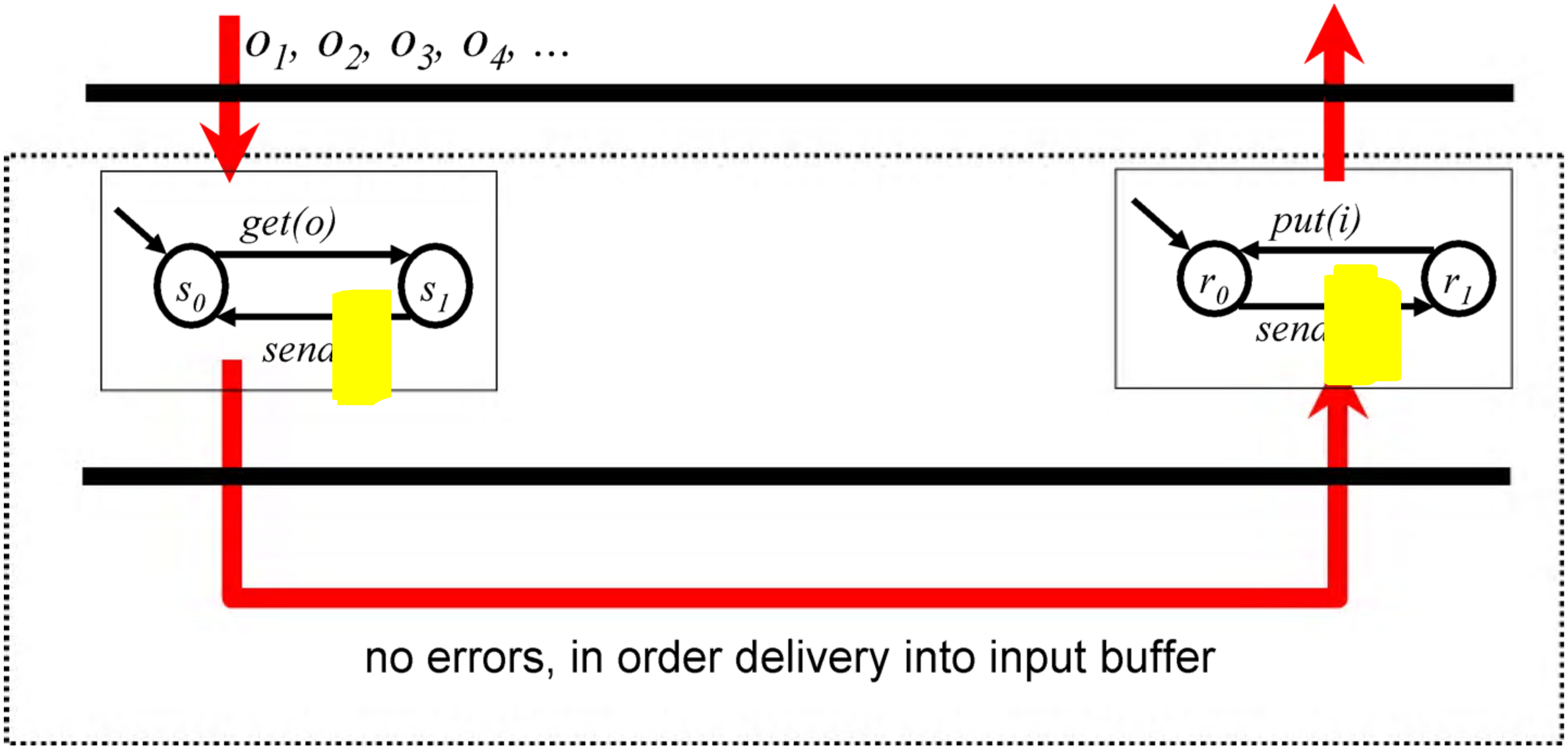


Recall: Protocol, Environment and Services



Getting started: An unrestricted simplex protocol

Getting started: An unrestricted simplex protocol



no errors, in order delivery into input buffer

The same in C:

```
/* Protocol 1 (utopia) provides for data transmission in one direction only, from
   sender to receiver.  The communication channel is assumed to be error free
   and the receiver is assumed to be able to process all the input infinitely quickly.
   Consequently, the sender just sits in a loop pumping data out onto the line as
   fast as it can. */

typedef enum {frame_arrival} event_type;
#include "protocol.h"
```


```
void sender1(void)
{
  frame s;                           /* buffer for an outbound fr
  packet buffer;                     /* buffer for an outbound p

  while (true) {
      from_network_layer(&buffer); /* go get something to sen
      s.info = buffer;             /* copy it into s for transmis
      to_physical_layer(&s);       /* send it on its way */
  }                                /* Tomorrow, and tomorrov
                                      Creeps in this petty pace
                                      To the last syllable of rec
                                         - Macbeth, V, v */
}
```

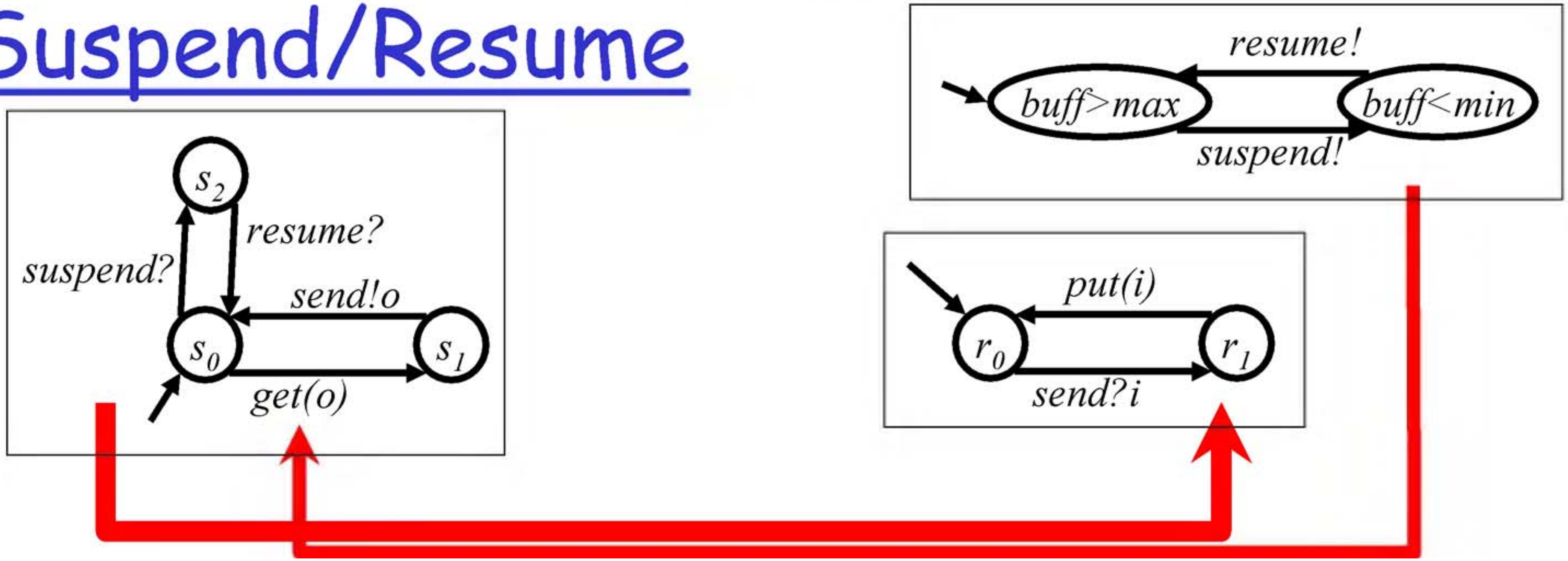
```
void receiver1(void)
{
  frame r;
  event_type event;                  /* filled in by wait, but not used here */

  while (true) {
      wait_for_event(&event);        /* only possibility is frame_arrival */
      from_physical_layer(&r);       /* go get the inbound frame */
      to_network_layer(&r.info);     /* pass the data to the network layer */
  }
}
```

Assessment:

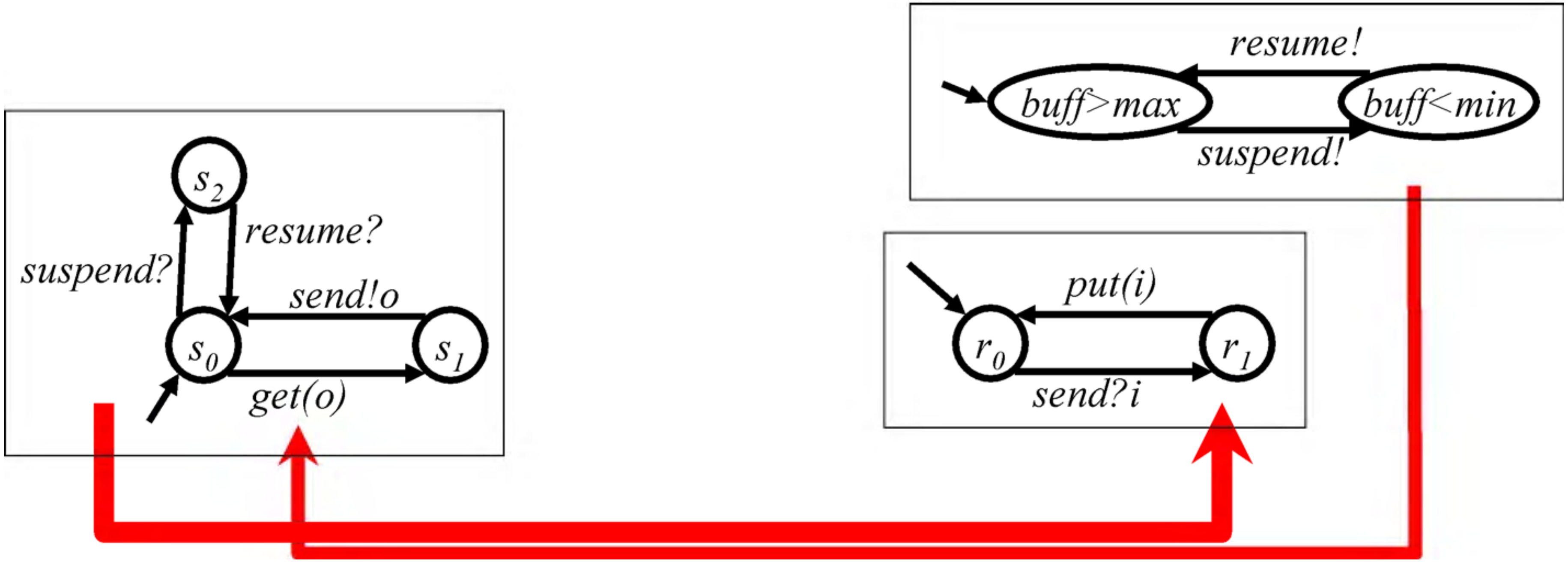
- Works.
- Sender may pump out data as fast as possible.
- Receiver must be assumed to work infinitely fast, to prevent buffer overflow.
- Unrealistic!

1st Revision: Simplex Suspend/Resume



- Unidirectional data-flow, but counterdirectional control-flow.
- Receiver now has finite buffer and processing speed, gives feedback - if buffer space runs out.
- Channel is still assumed error-free.

Assessment:



- Delay of 'suspend' messages needs to be predictable, to prevent buffer overflow.
- Unrealistic!

(Loss of resume message would make the system deadlock.)

2nd Revision: Simplex stop-and-wait

- Again, unidirectional data-flow, but counterdirectional control-flow.
- Receiver has finite buffer and processing speed, gives feedback - once data is processed,
- Sender awaits ACKs before sending next data item.
- Channel is still assumed error-free.

The same in C:


```
void receiver2(void)
{
  frame r, s;                        /* buffers for frames */
  event_type event;                  /* frame_arrival is the only possibility */
  while (true) {
       wait_for_event(&event);       /* only possibility is frame_arrival */
       from_physical_layer(&r);      /* go get the inbound frame */
       to_network_layer(&r.info);    /* pass the data to the network layer */
       to_physical_layer(&s);        /* send a dummy frame to awaken sender */
  }
```

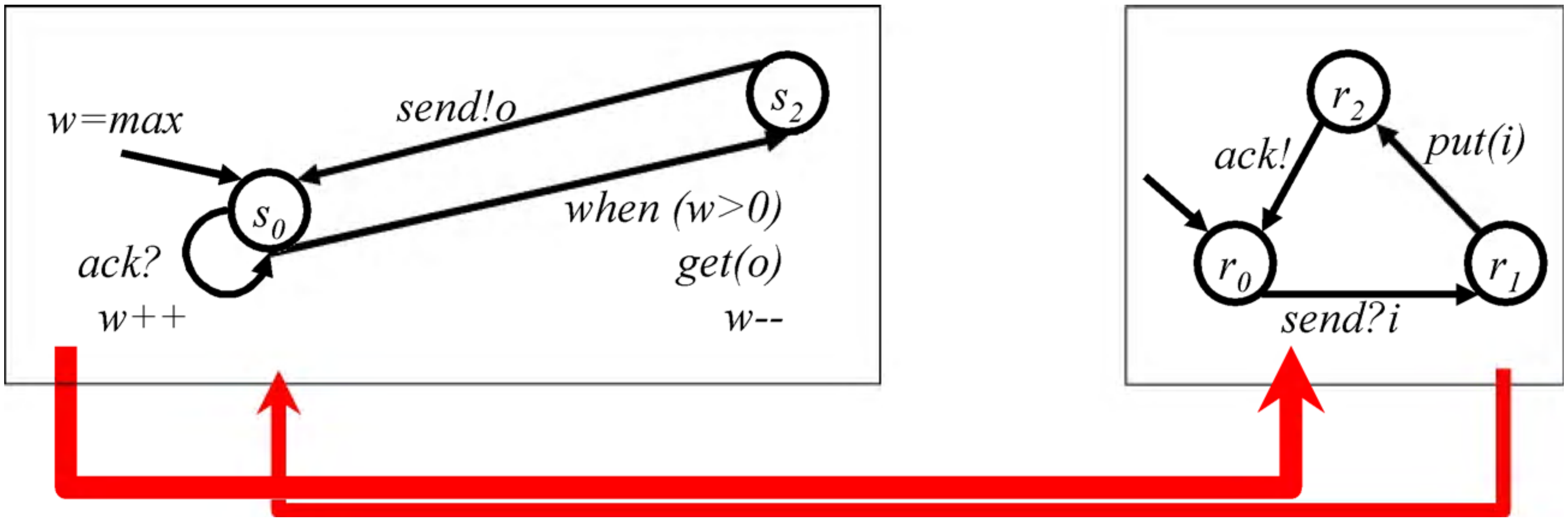
```
void sender2(void)
{
  frame s;                           /* buffer for an outboun
  packet buffer;                     /* buffer for an outboun
  event_type event;                  /* frame_arrival is the c

  while (true) {
       from_network_layer(&buffer);  /* go get something to send */
       s.info = buffer;              /* copy it into s for transmission */
       to_physical_layer(&s);        /* bye-bye little frame */
       wait_for_event(&event);       /* do not proceed until given the go ahead */
  }
}
```

Assessment:

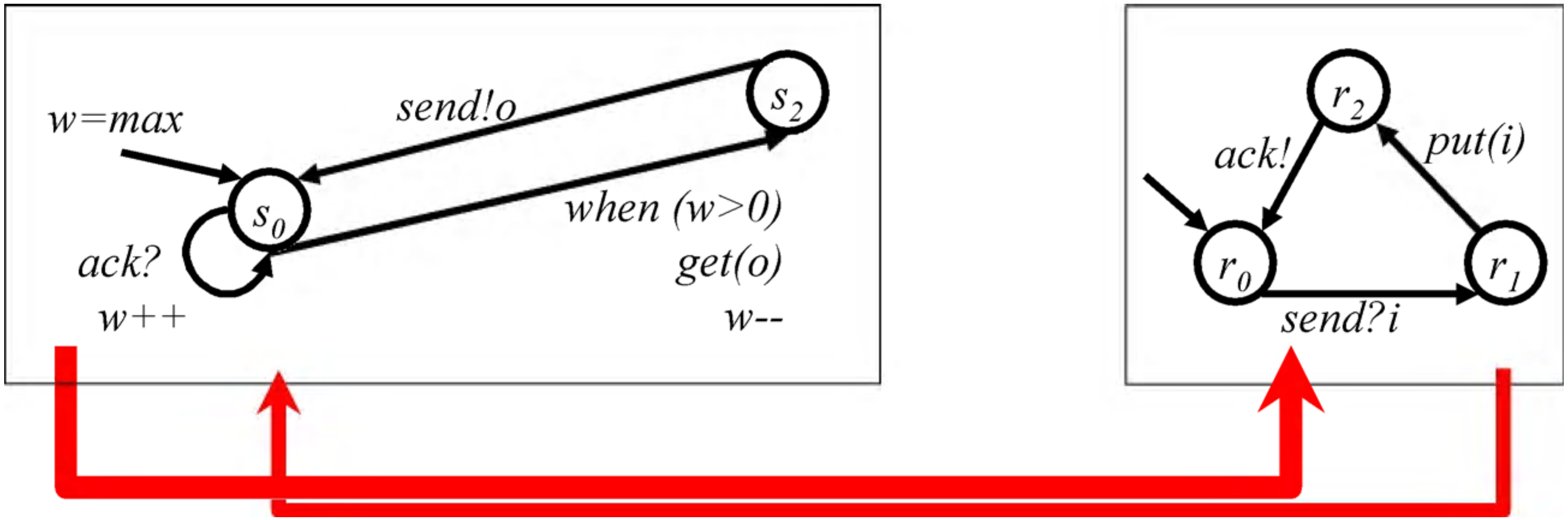
- Works fine, though not very efficient, under our assumption of a perfect channel.

3rd revision: adding a window



- Now the ACKs are decoupled from the sends. Sender maintains a worst-case estimate of the buffer space available at receiver.
- 'max' must be communicated from receiver to sender in initialisation phase.

Assessment:



- Works fine,
  under our assumption of a perfect channel.

- This is a basic flow-control mechanism.

(Unfortunately the perfect channel assumption is unrealistic.)

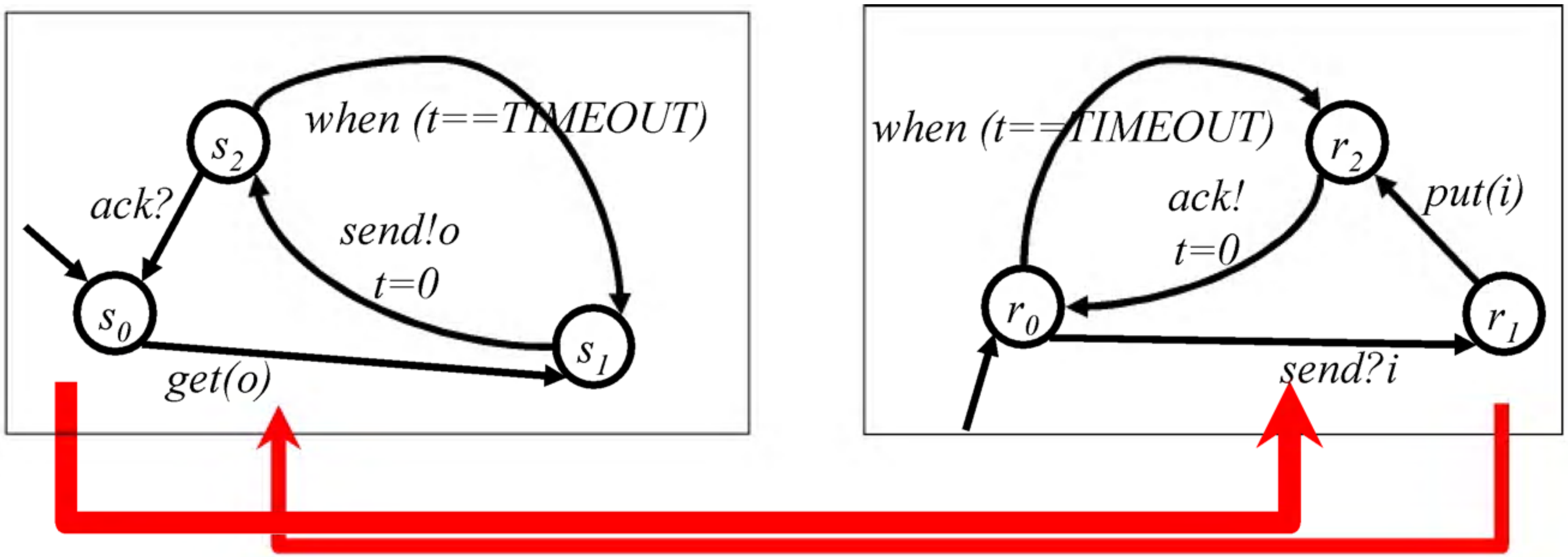
Noisy channels

Getting started:

- A noisy channel may corrupt messages, or *lose messages without any notification*
  (different from Session A, where silent losses were not considered)

- In the presence of potential losses, what to do?
  - Wait, wait, wait?
  - Send packets multiple times? When?
- The principal way out: Timers and timeouts.

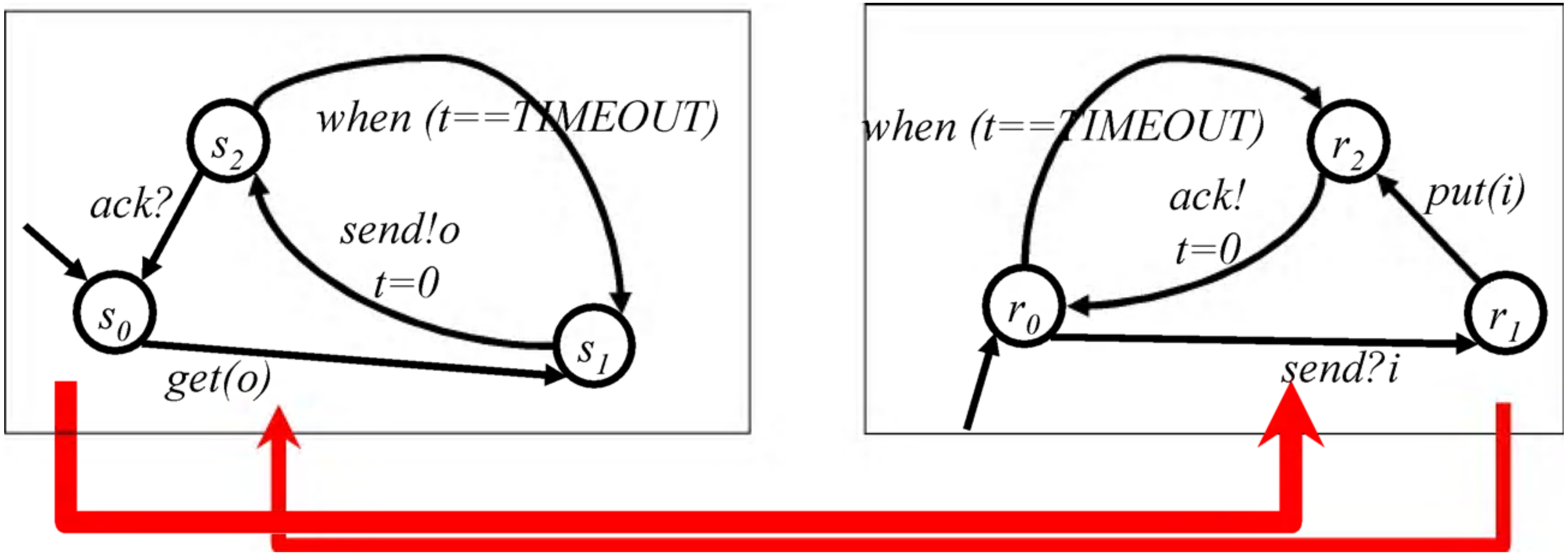
4th Revision: Simplex stop-and-wait over noisy channel



Assumption: Channel may corrupt messages, or lose messages without any notification.

Corrupted message are simply discarded as if they were lost.

Assessment:



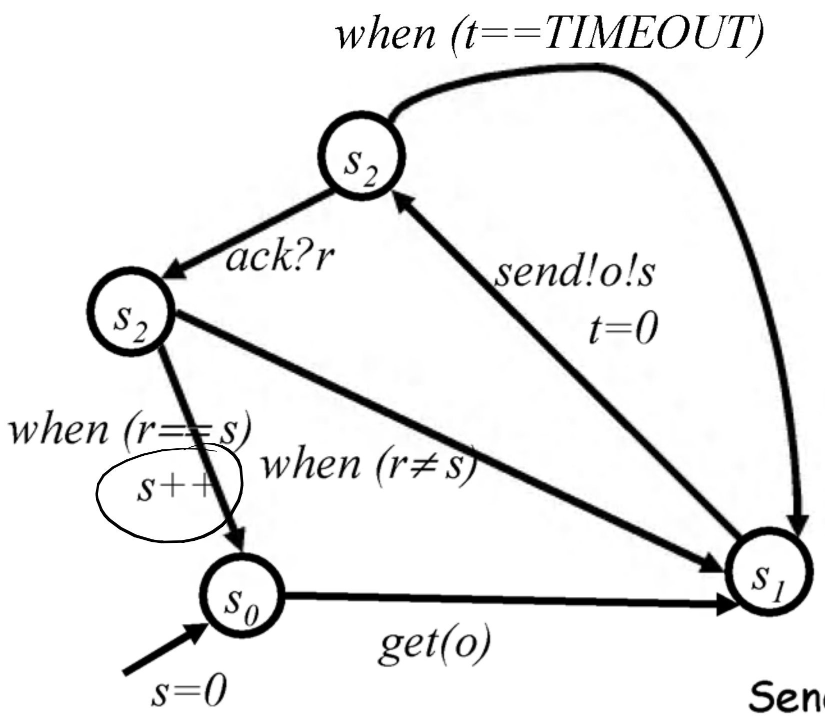
- It is still too simplistic and does not work. Think about it.
- The problem is that data may be duplicated because of lost ACKs.
- Hmhh. What is needed is a way to tell whether some data has been seen before.

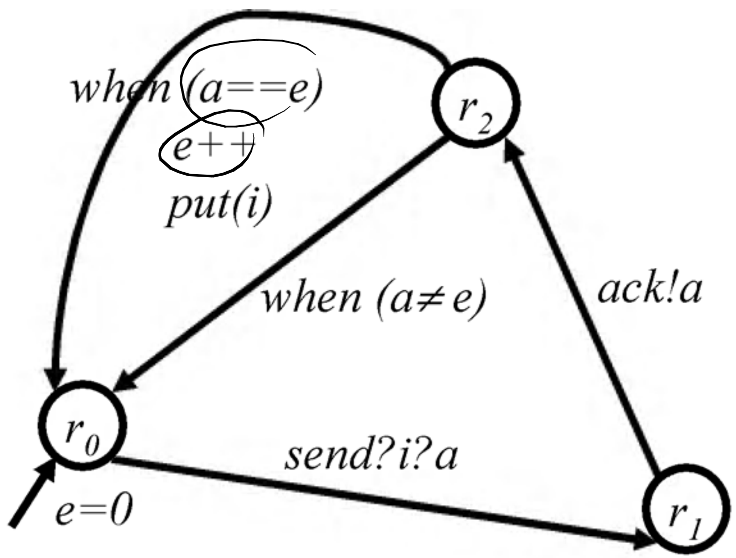
'sequence numbers'

Sequence numbers

- The sender maintains a counter for the l[illegible]ge it has been sending, and waits for an [illegible]r this message.
- Similar for receiver.
- The se[illegible]with the message.
- The receiver echoes th[illegible] the [illegible]
- Both check whether th[illegible] was expected.

5th Revision: Simplex stop-and-wait with sequence numbers





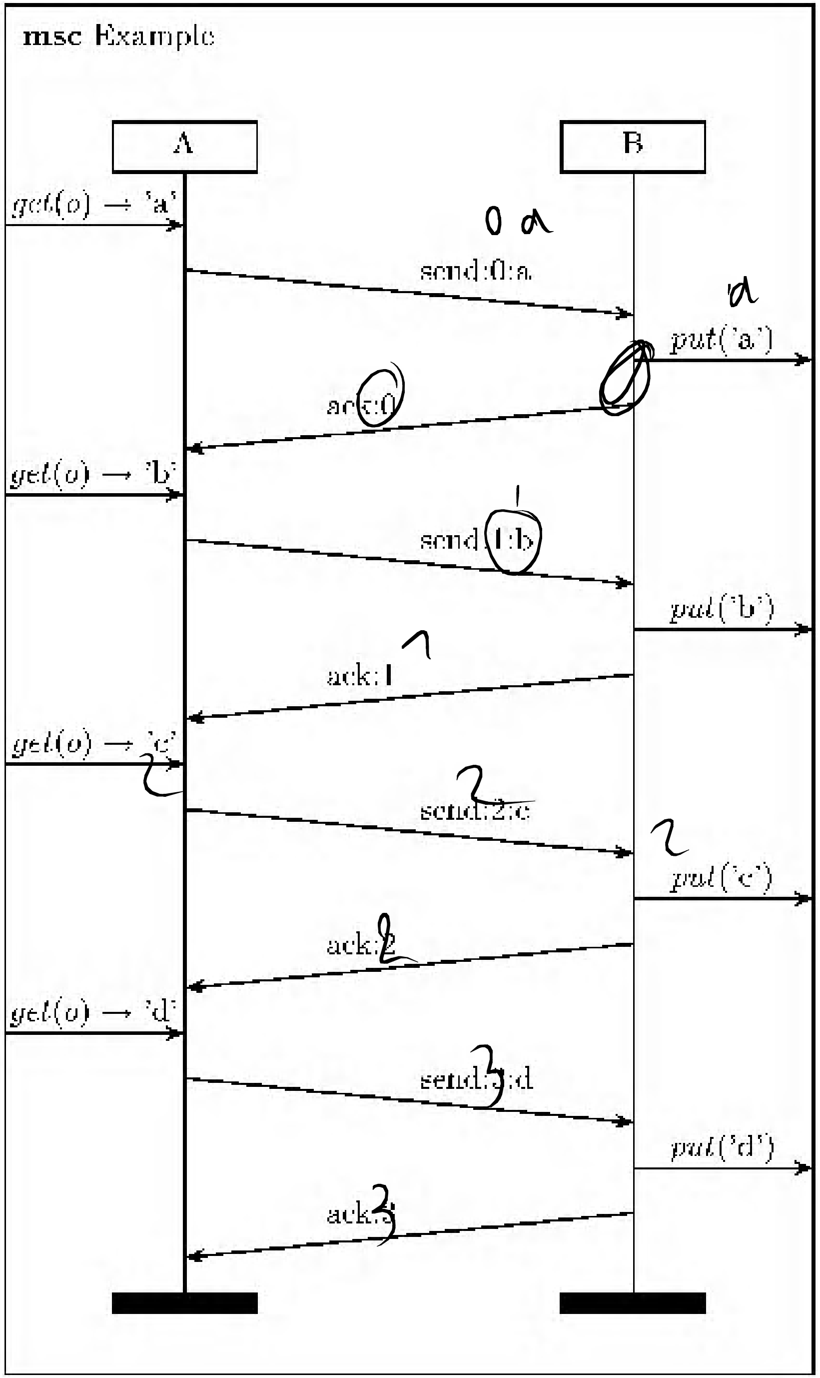
Sender
s: last # sent
r: last # received

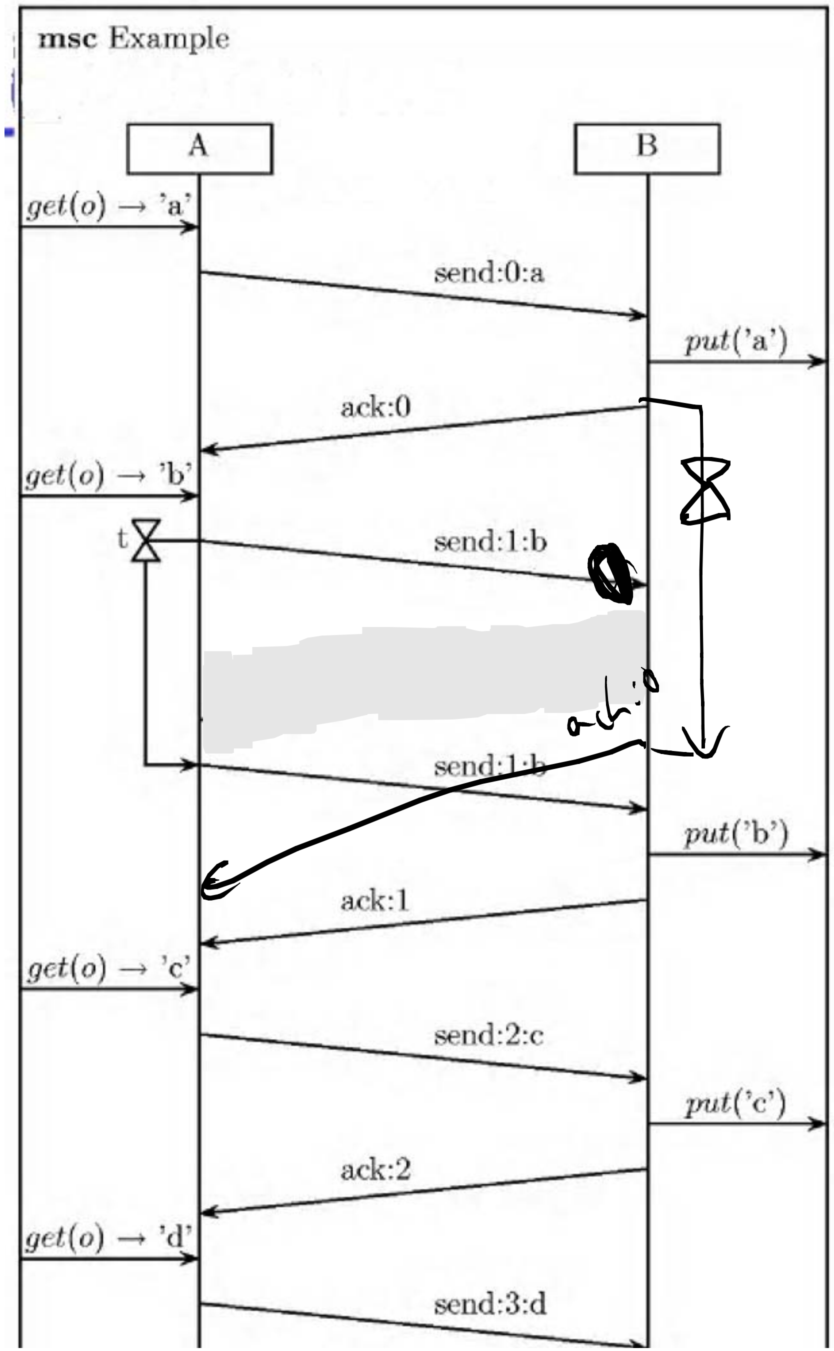
Receiver
e: next # expected
a: last # received

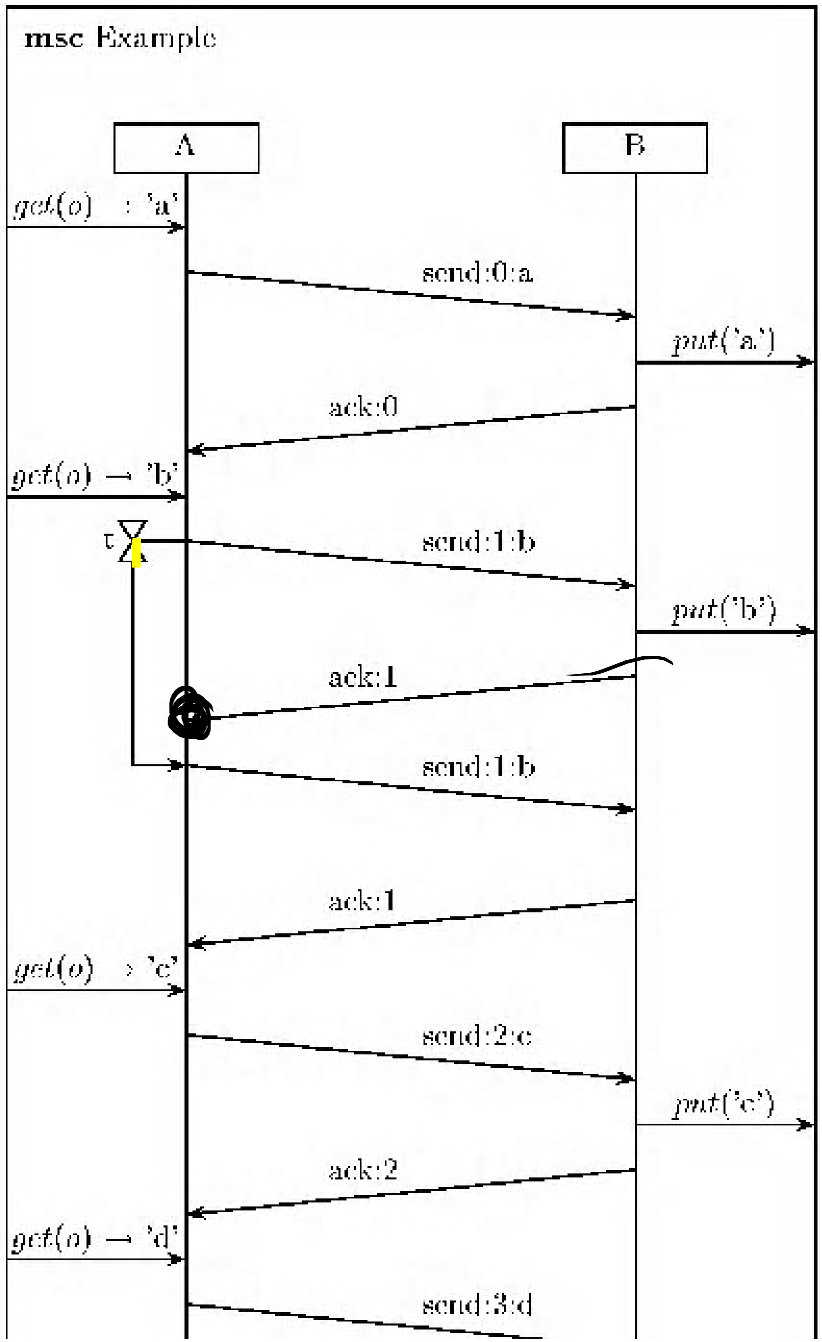
Corrupted message are simply discarded as if they were lost.

Examples

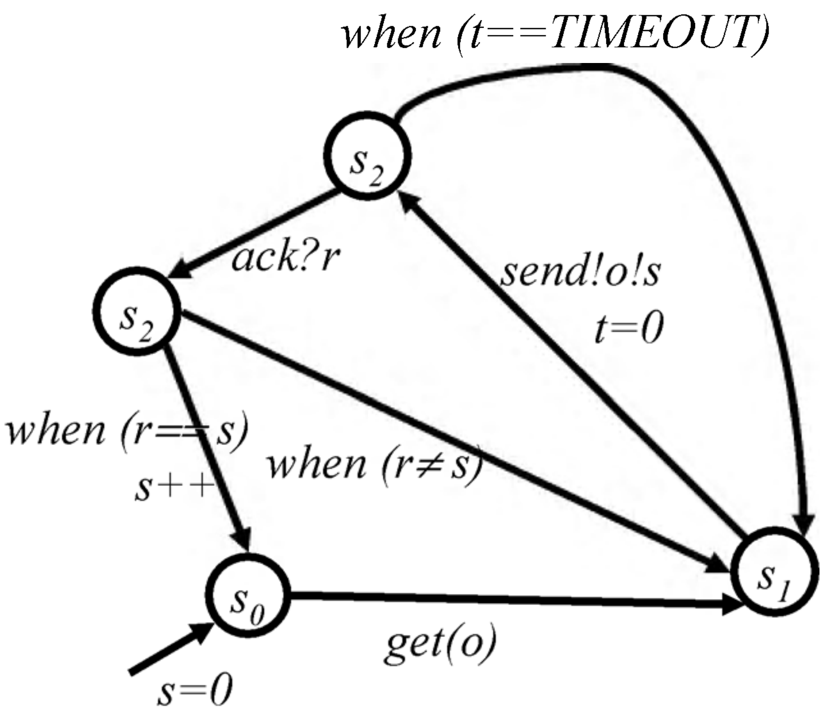
abc...xyz

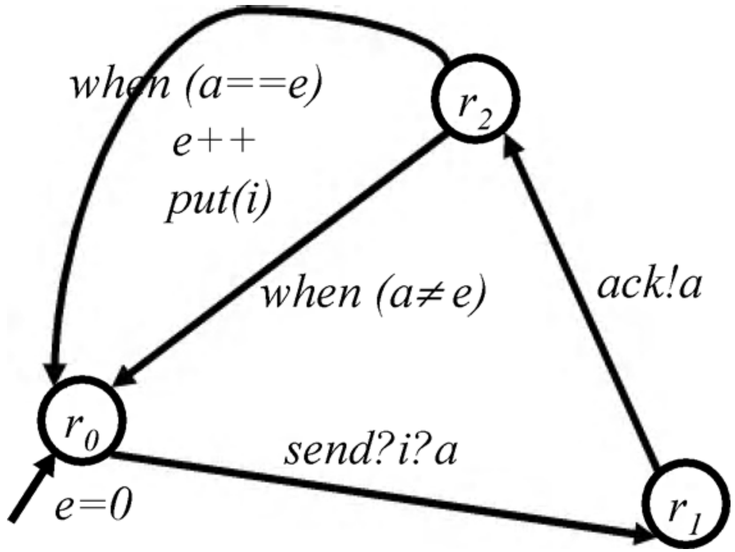



msc Example
A
B
get(o) → 'a'
send:0:a
put('a')
ack:0
get(o) → 'b'
t
send:1:b
ack:0
send:1:b
put('b')
ack:1
get(o) → 'c'
send:2:c
put('c')
ack:2
get(o) → 'd'
send:3:d


msc Example
A
B
get(o) → 'a'
send:0:a
put('a')
ack:0
get(o) → 'b'
t
send:1:b
put('b')
ack:1
send:1:b
ack:1
get(o) → 'c'
send:2:c
put('c')
ack:2
get(o) → 'd'
send:3:d

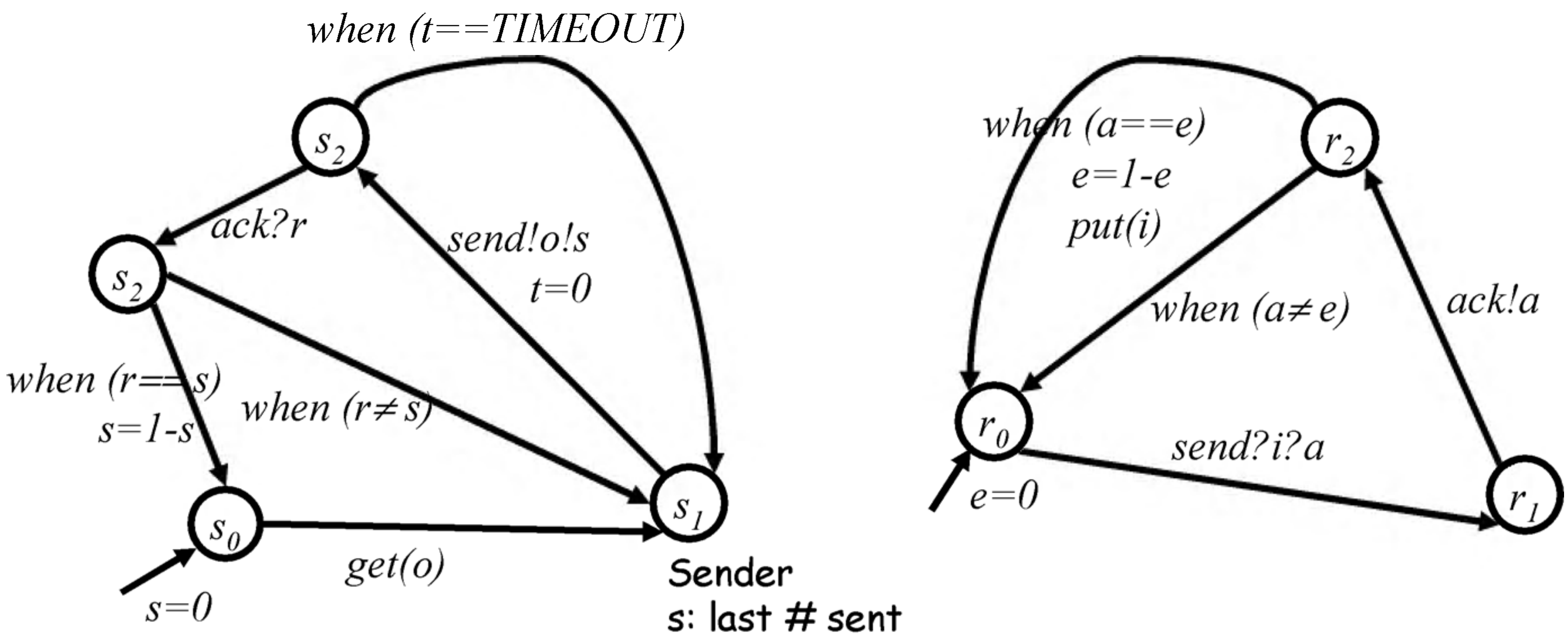
Assessment:





- Believe it, or not. But this one works.
- Still it is a little suboptimal, because the sequence number grow, and thus eat up transmission capacity.

6th Revision: Just two sequence numbers



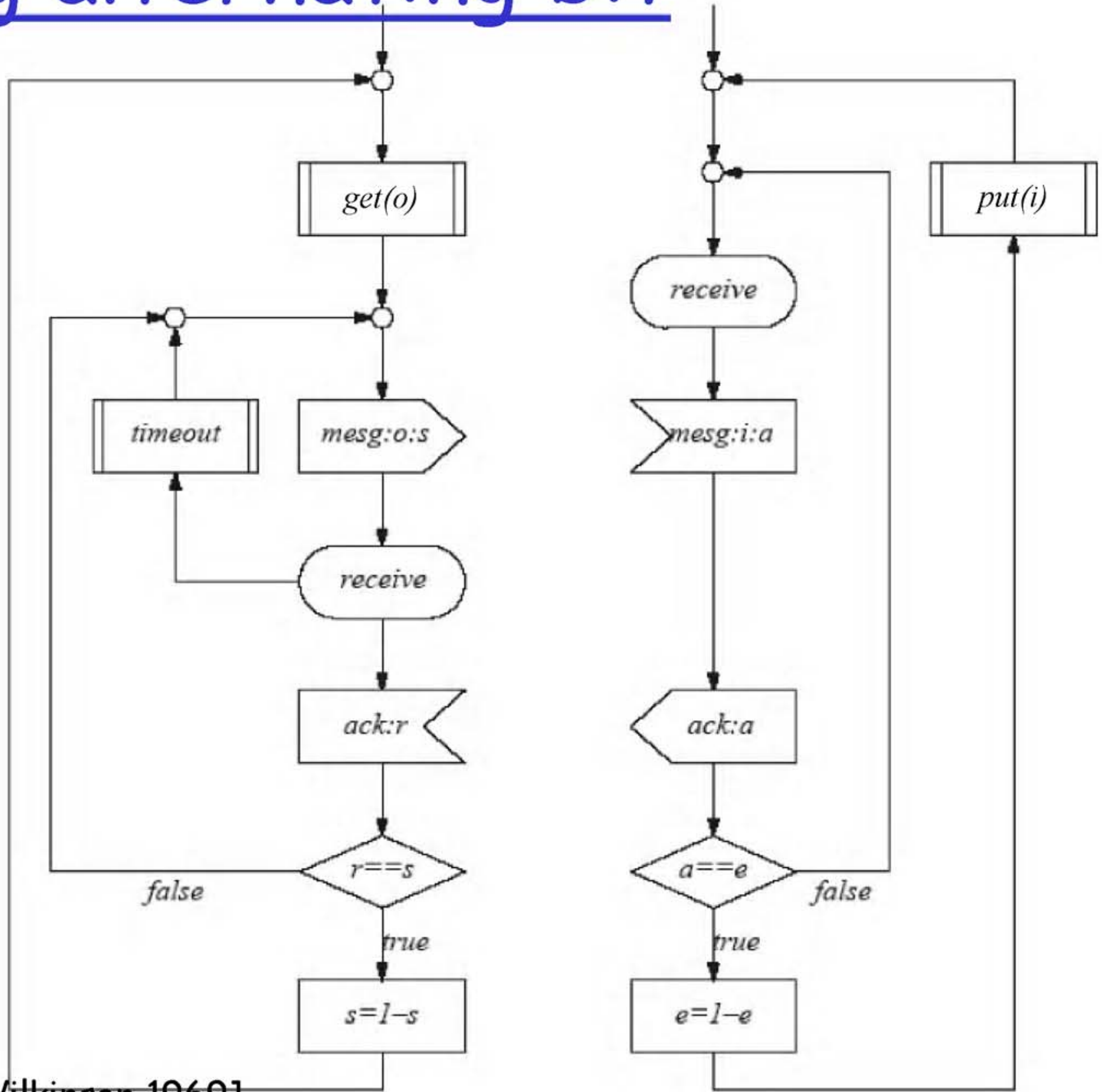
Sender
s: last # sent
r: last # received

Receiver
e: next # expected
a: last # received

- This one alternates a single bit, and still it works!

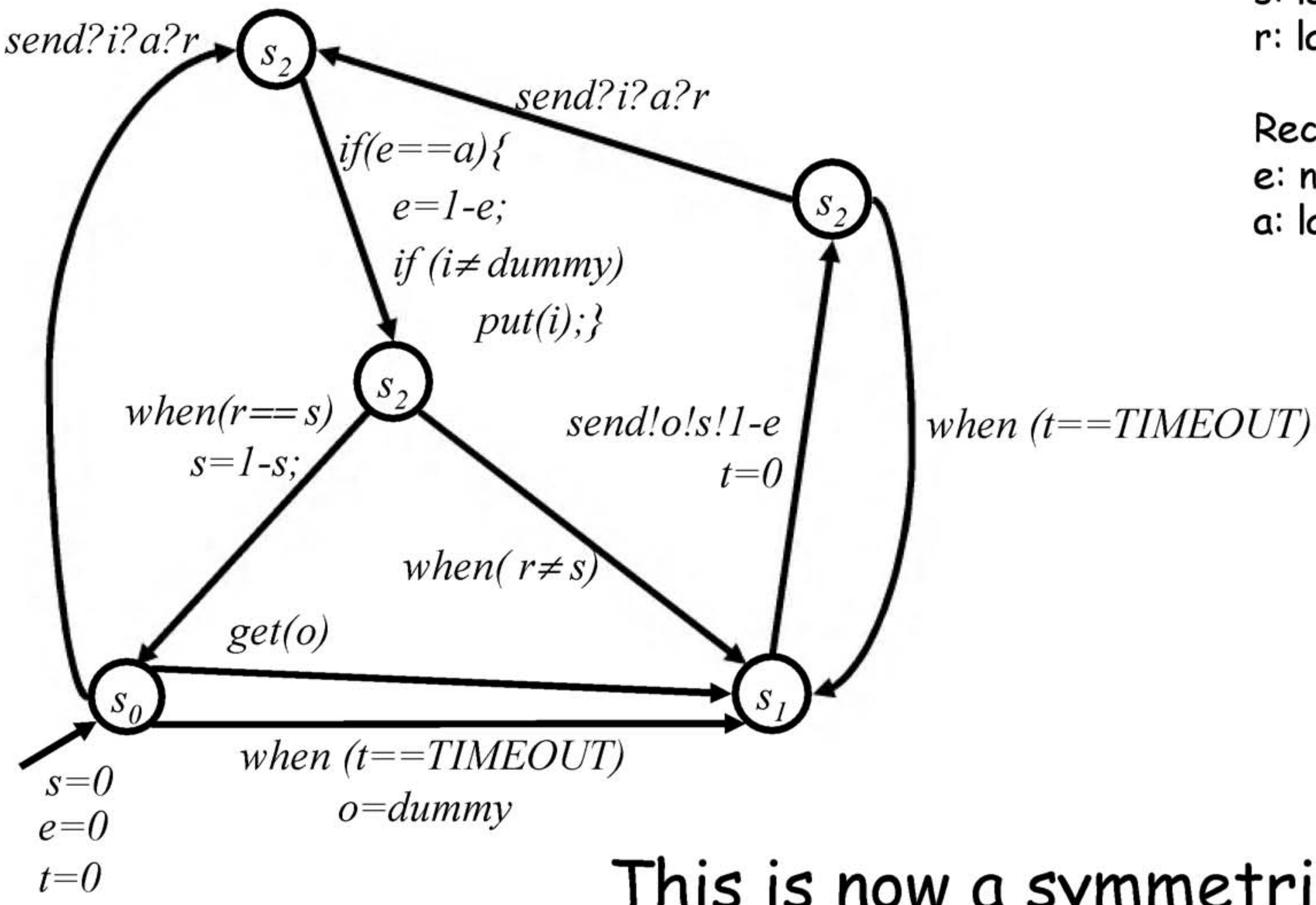
The amazing alternating bit protocol

A bullet-proof protocol.



[Bartlett, Scantlebury and Wilkinson 1969].

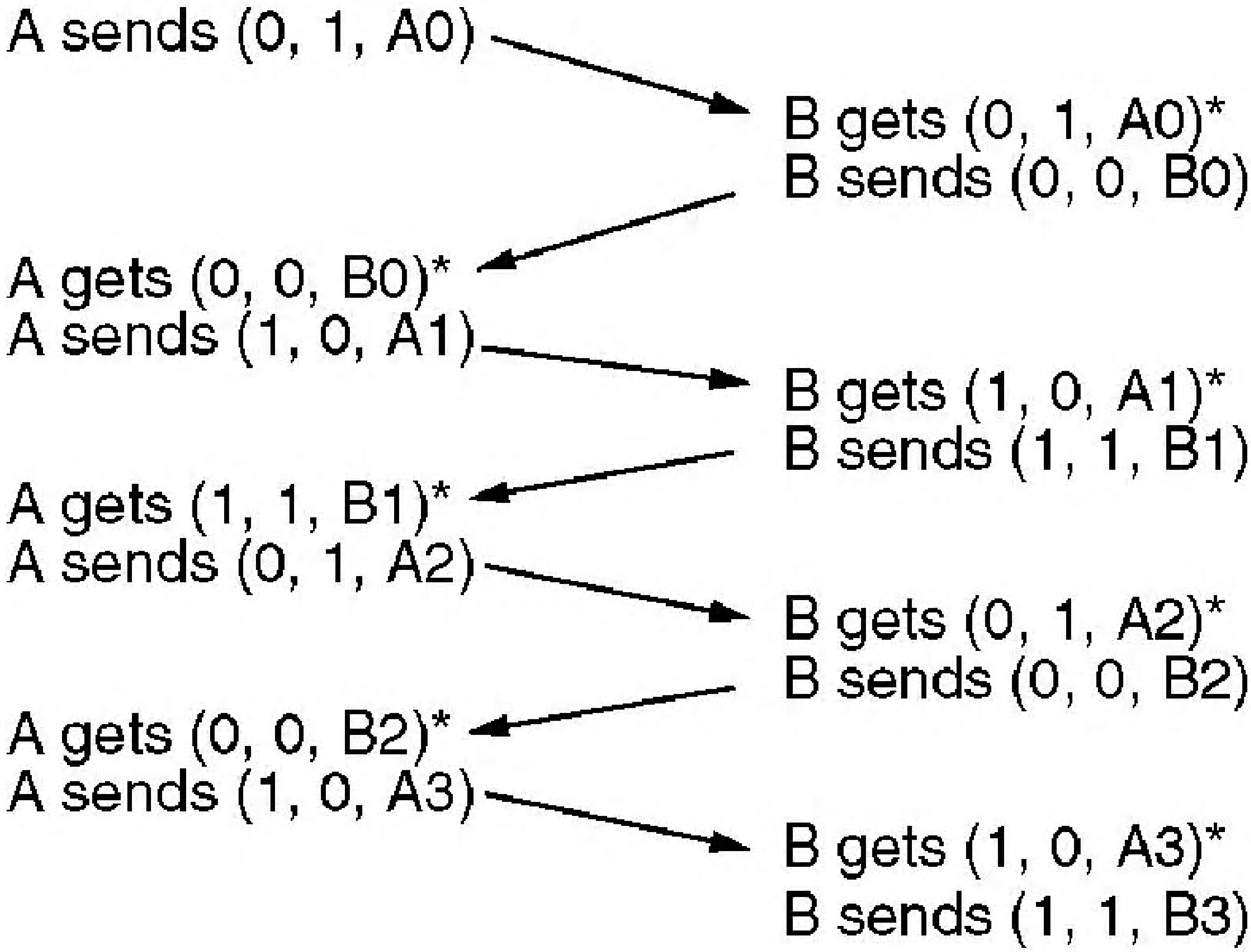
7th Revision: Alternating bit with full duplex and piggybacking

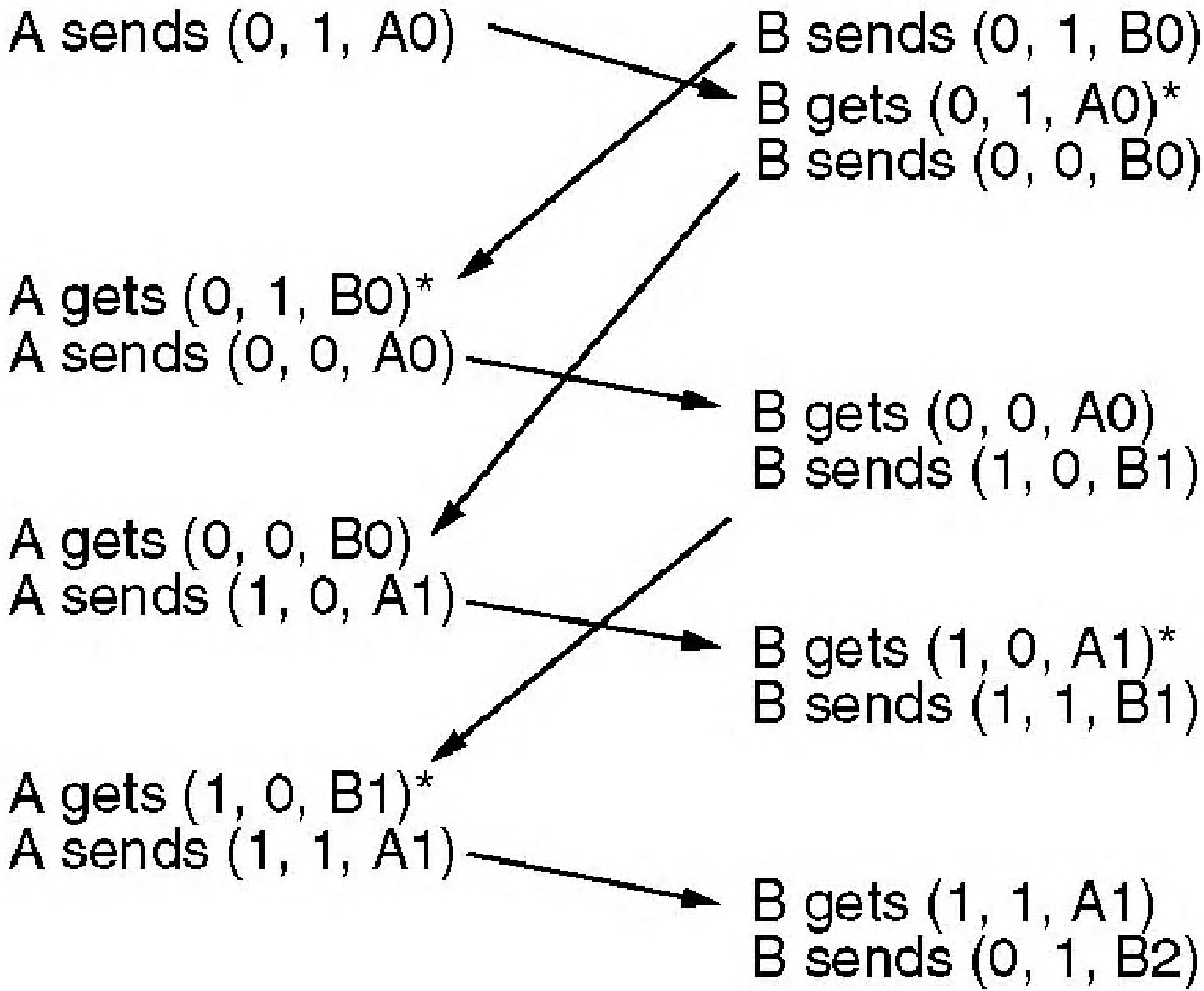


Sender part
s: last # sent
r: last # received

Receiver part
e: next # expected
a: last # received

This is now a symmetric protocol.


A sends (0, 1, A0)
B gets (0, 1, A0)*
B sends (0, 0, B0)
A gets (0, 0, B0)*
A sends (1, 0, A1)
B gets (1, 0, A1)*
B sends (1, 1, B1)
A gets (1, 1, B1)*
A sends (0, 1, A2)
B gets (0, 1, A2)*
B sends (0, 0, B2)
A gets (0, 0, B2)*
A sends (1, 0, A3)
B gets (1, 0, A3)*
B sends (1, 1, B3)


A sends (0, 1, A0)
B sends (0, 1, B0)
B gets (0, 1, A0)*
B sends (0, 0, B0)
A gets (0, 1, B0)*
A sends (0, 0, A0)
B gets (0, 0, A0)
B sends (1, 0, B1)
A gets (0, 0, B0)
A sends (1, 0, A1)
B gets (1, 0, A1)*
B sends (1, 1, B1)
A gets (1, 0, B1)*
A sends (1, 1, A1)
B gets (1, 1, A1)
B sends (0, 1, B2)

Assessment:

- You assess it.

Networking

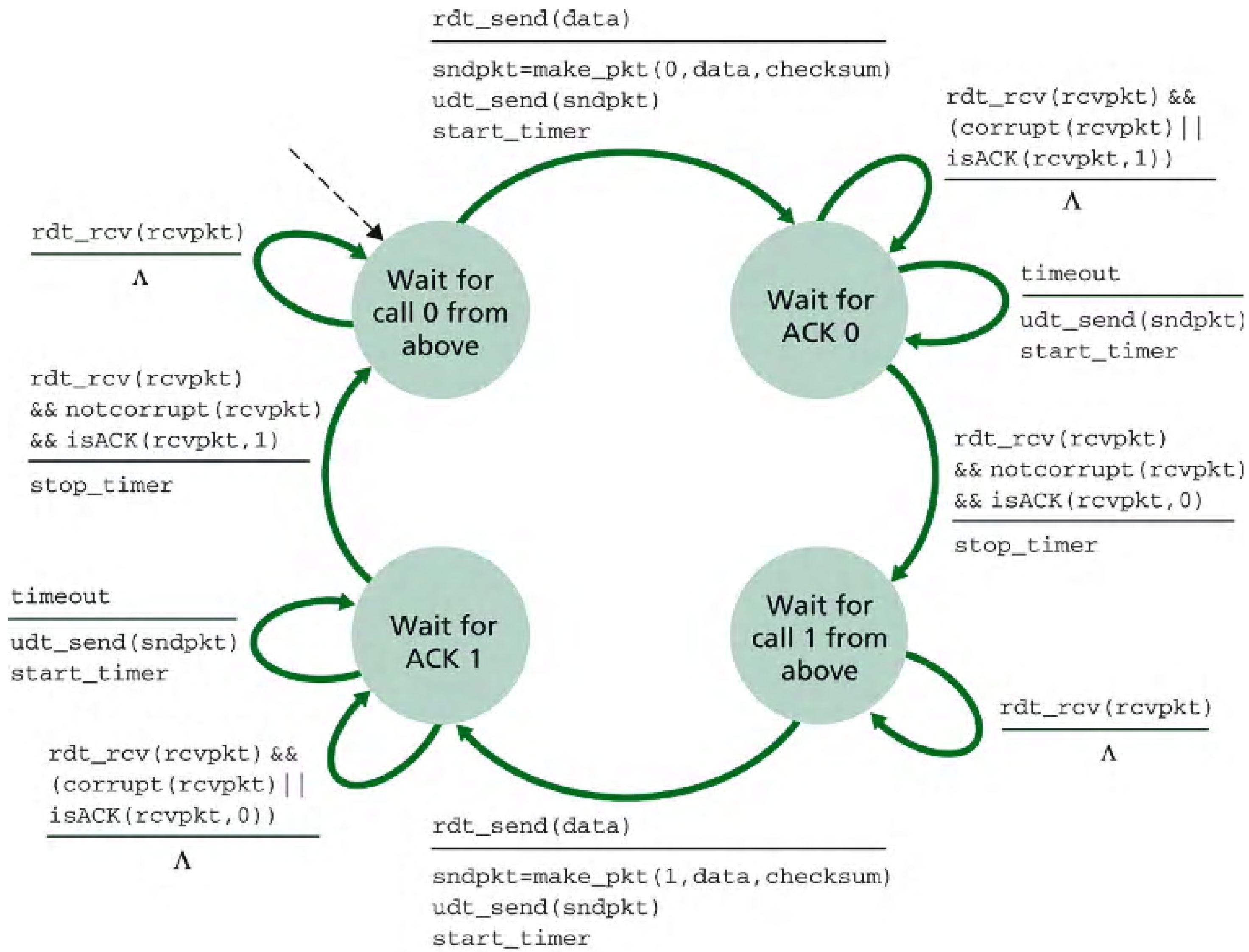
Prof. Dr.-Ing. Holger Hermanns

Dependable Systems & Software
Saarland University

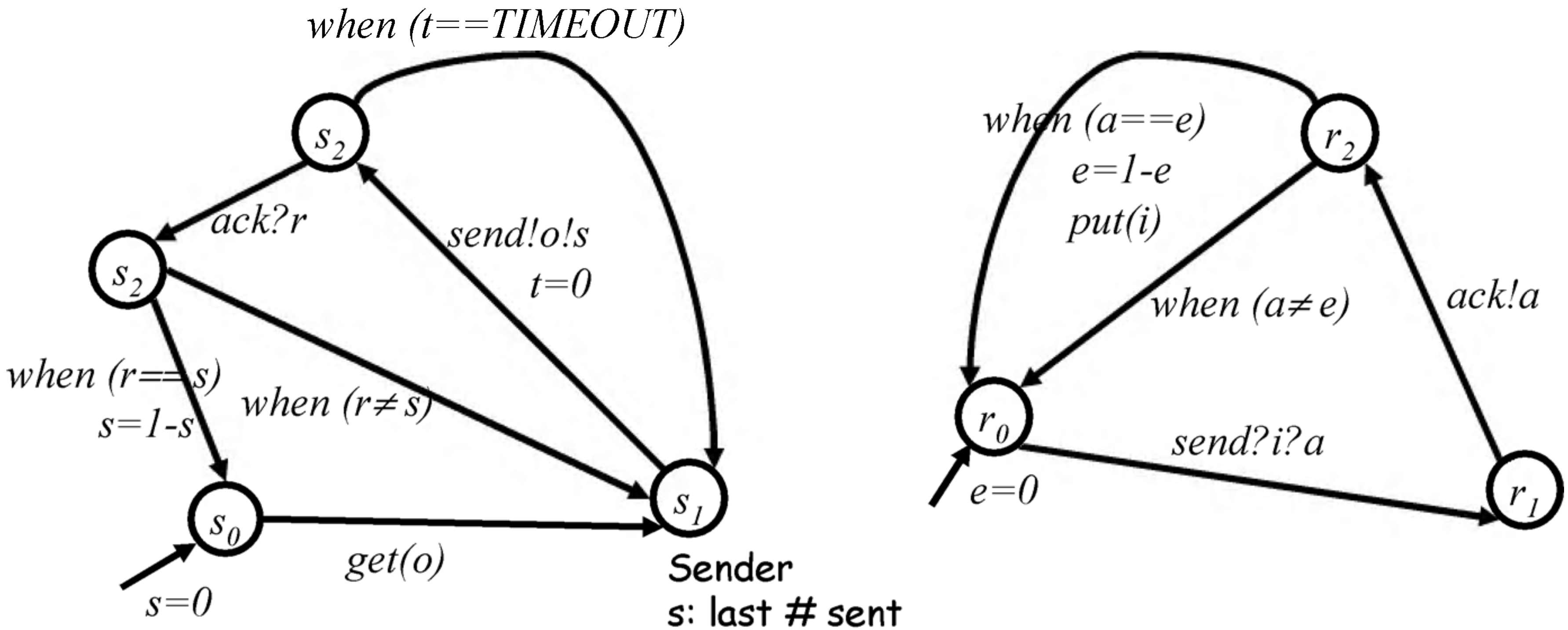
Summer 04

Lecture 5: Transport Layer

rdt3.0 sender of [Kurose Ross]: The Alternating Bit protocol



Recall the Alternating-Bit Protocol

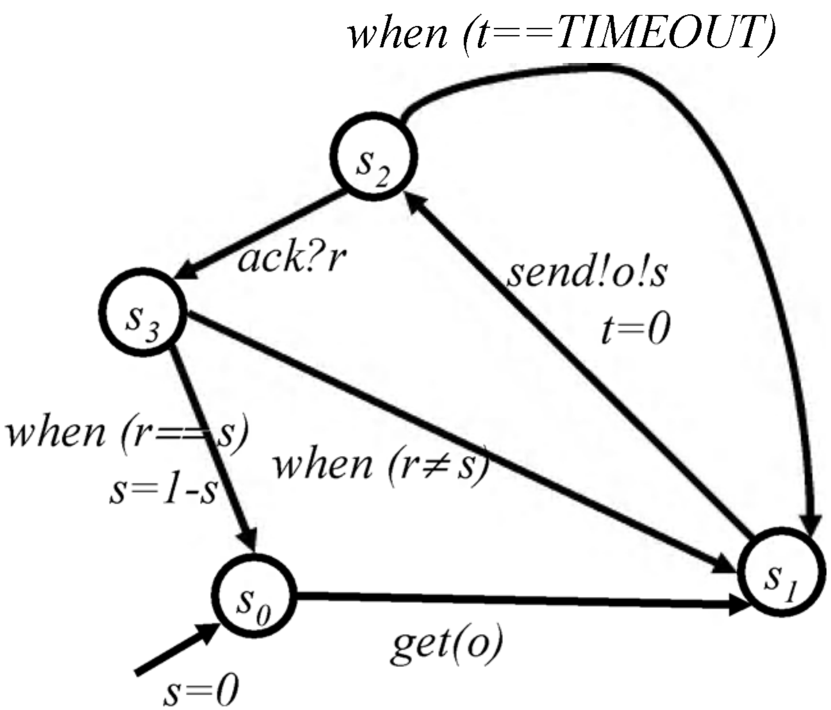


Sender
s: last # sent
r: last # received

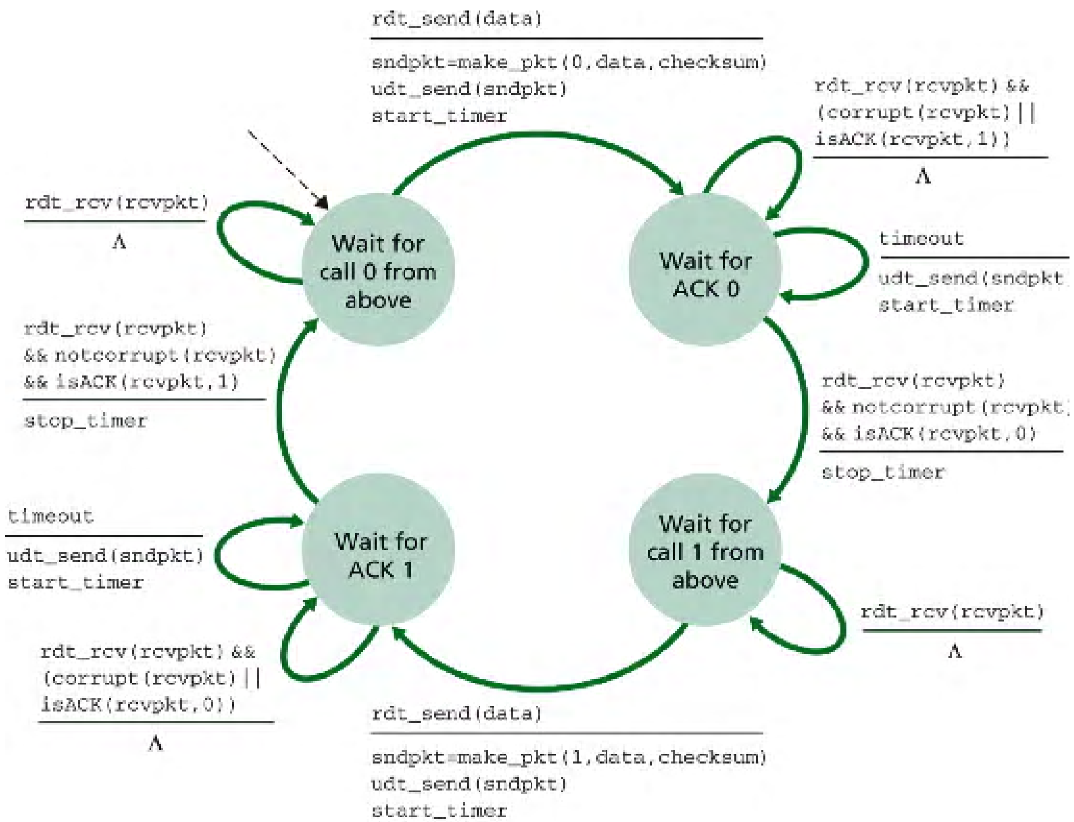
Receiver
e: next # expected
a: last # received

- This one alternates a single bit, and still it works!

Wait, wait!



Sender
s: last # sent
r: last # received



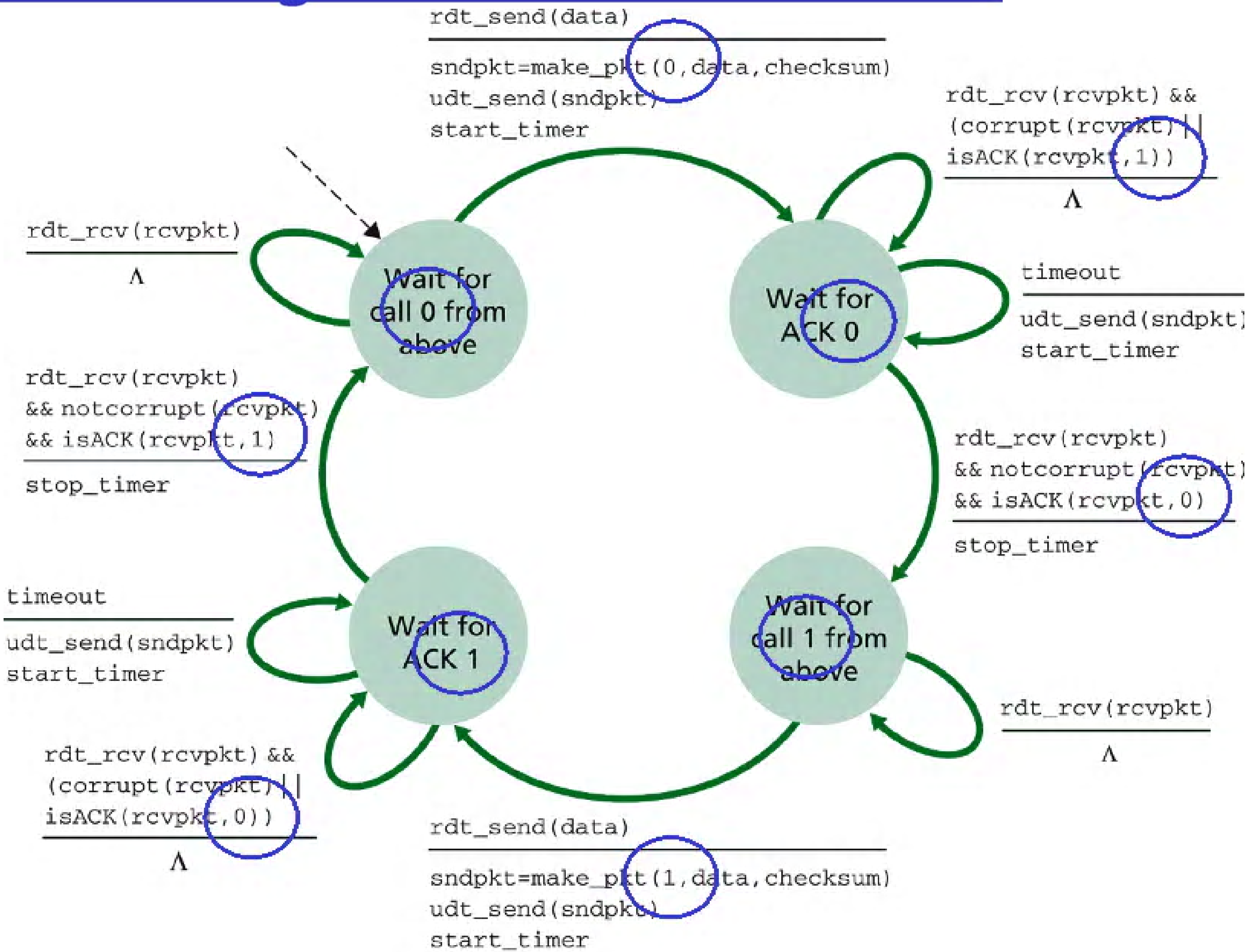
- Doesn't look like they describe the same protocol!

A step-by-step transformation

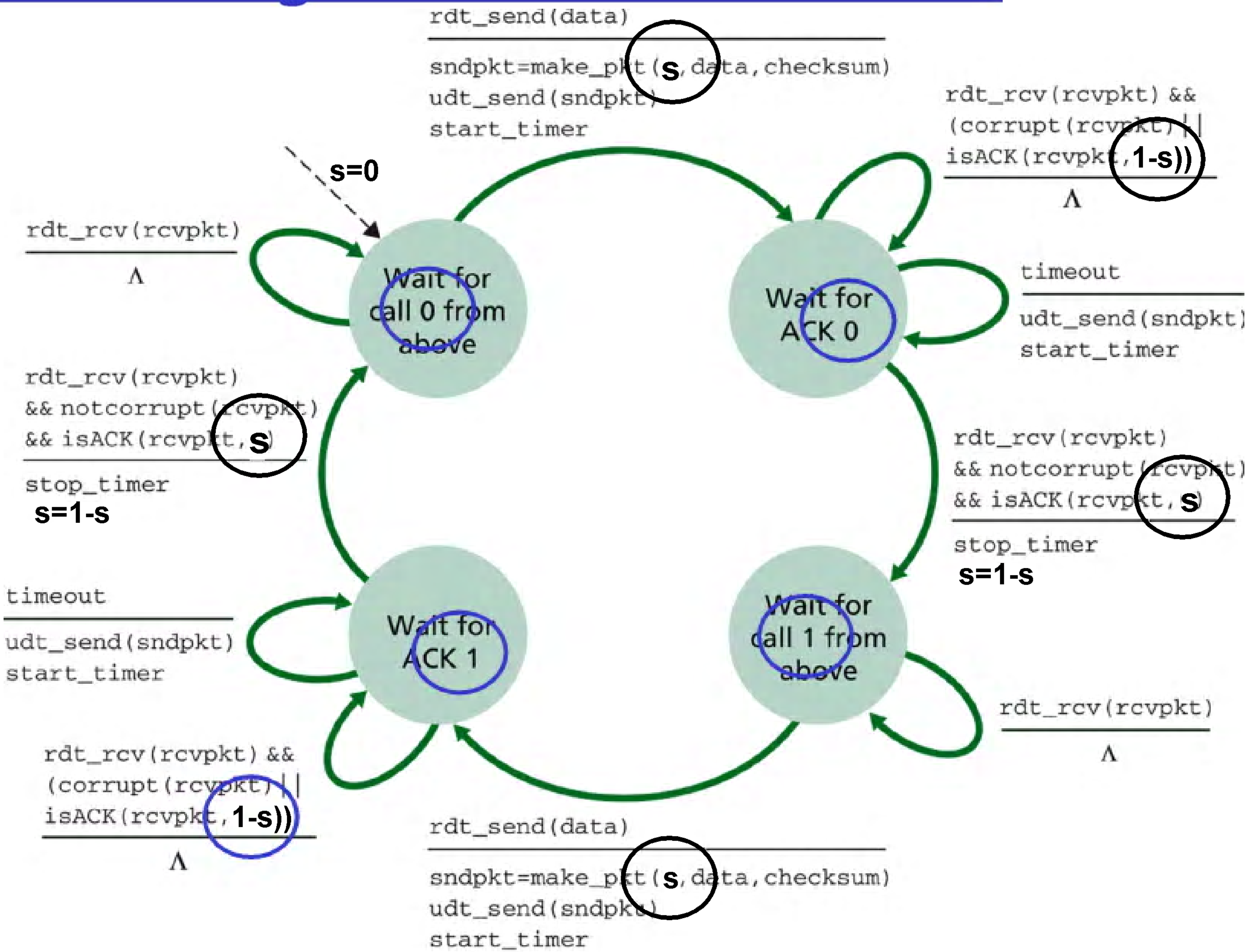
- We'll go through a sequence of transformation steps for the two state-transitition diagrams.
-
- Each step will be presented informally.
- A formal treatment requires a homomorphism from one diagram to the other.

- Homomorphism: property-preserving mapping
- here properties are: traces (e.g. sequence diagrams) generated by either specification (in interaction with the receiver side)
- for any trace generated in one mode, there should be a homomorphic trace in the other model, and vice versa.

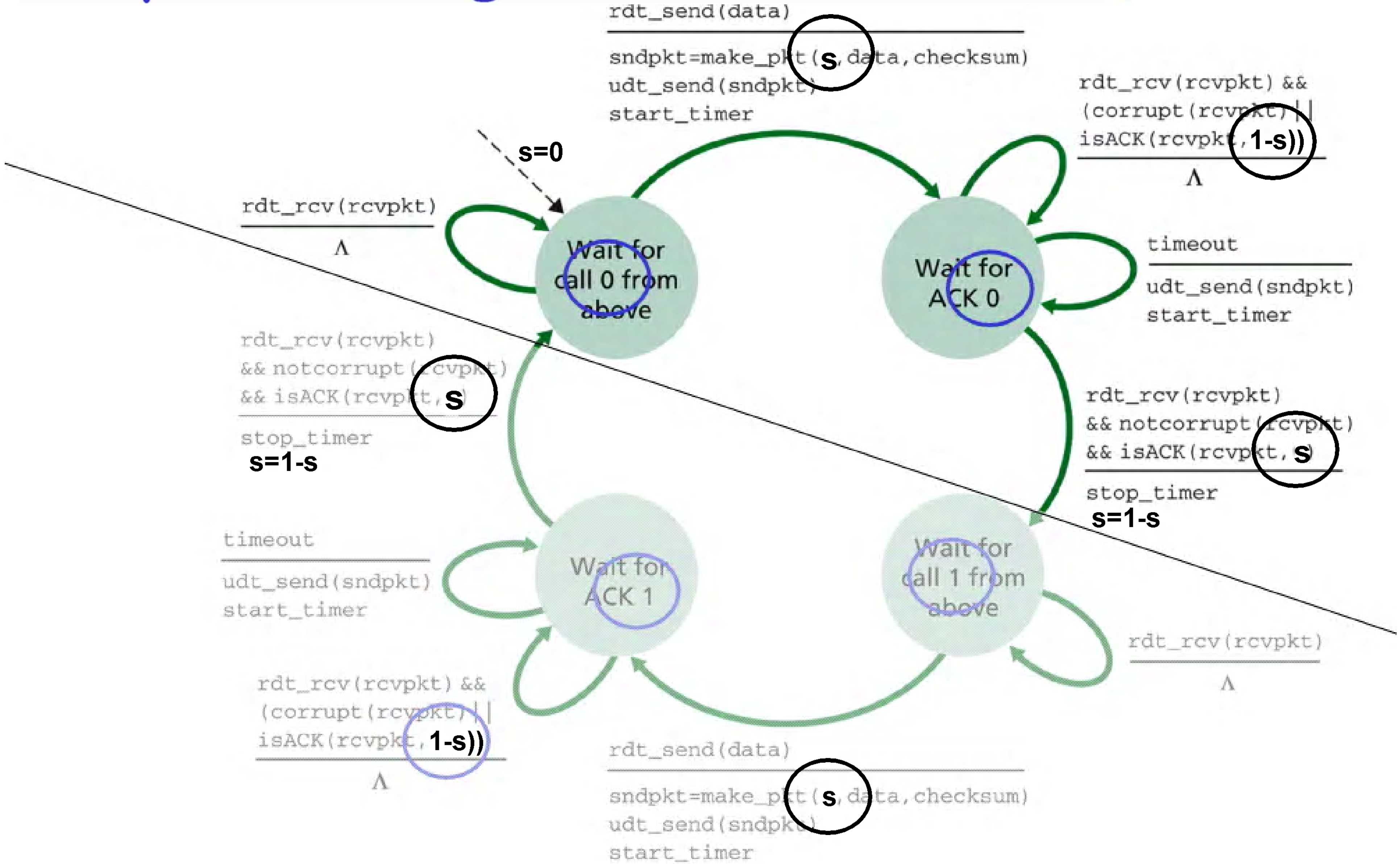
Step I: Folding the alternation bit



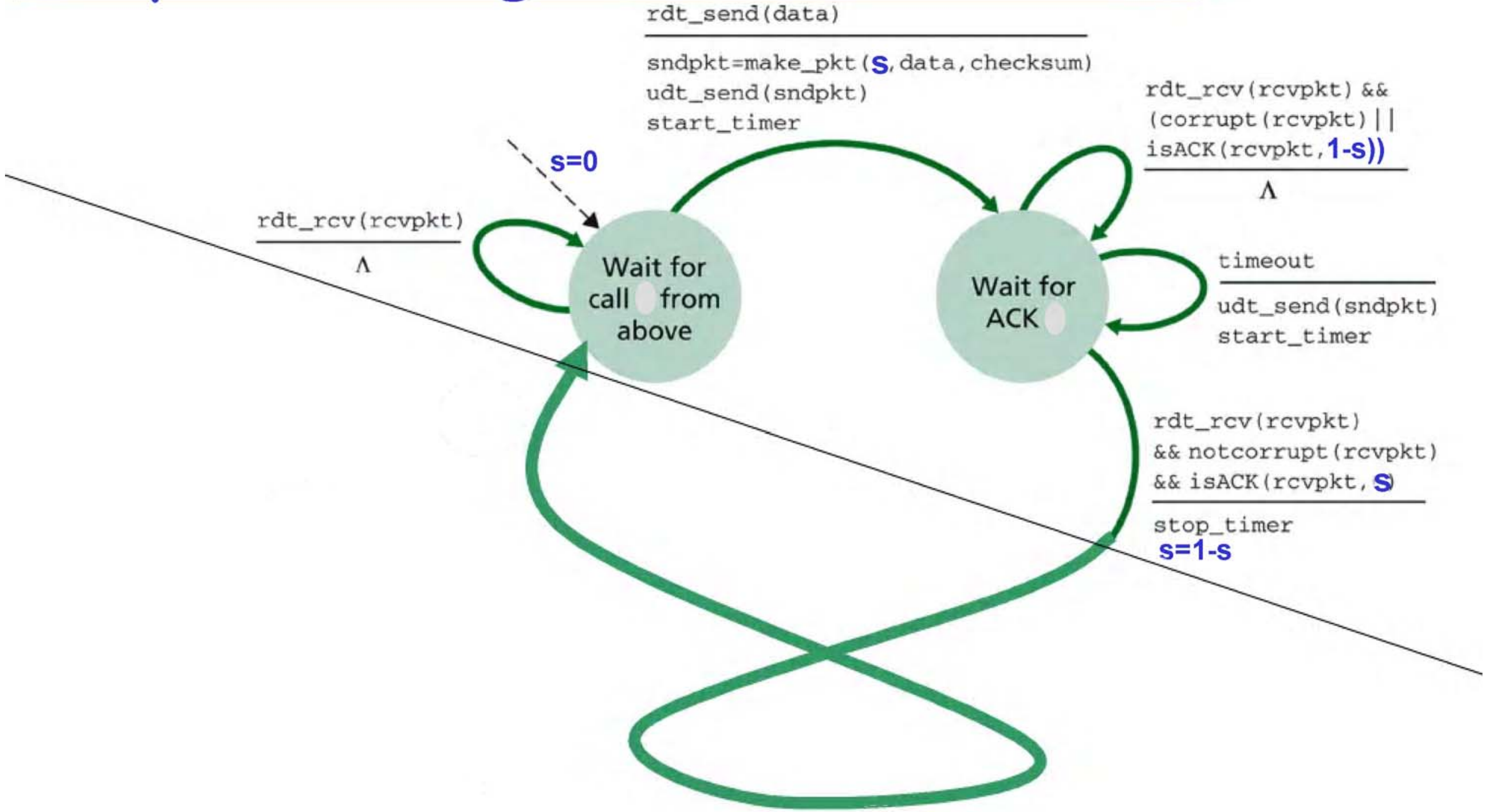
Step I: Folding the alternation bit



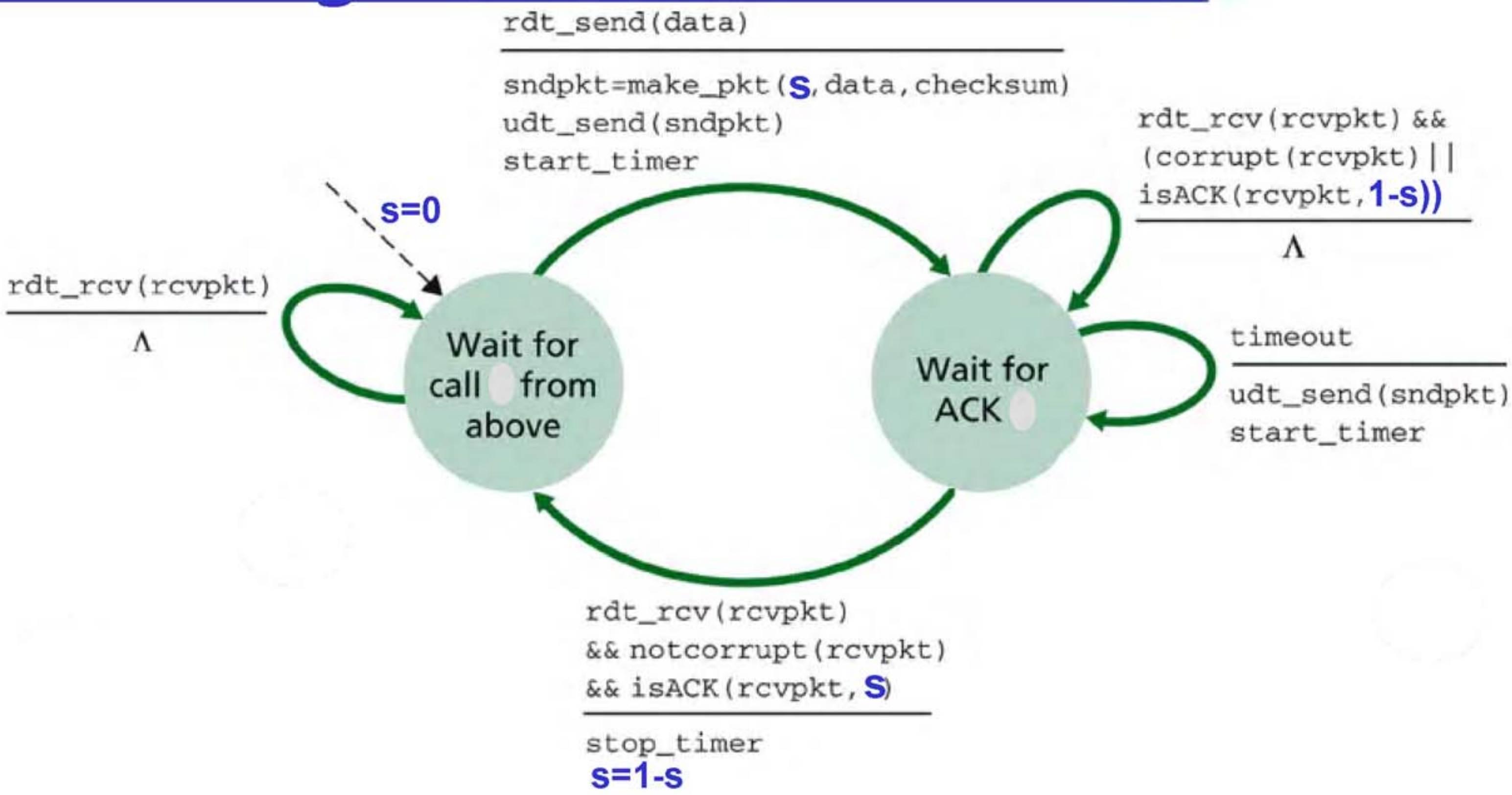
Step I: Folding the alternation bit



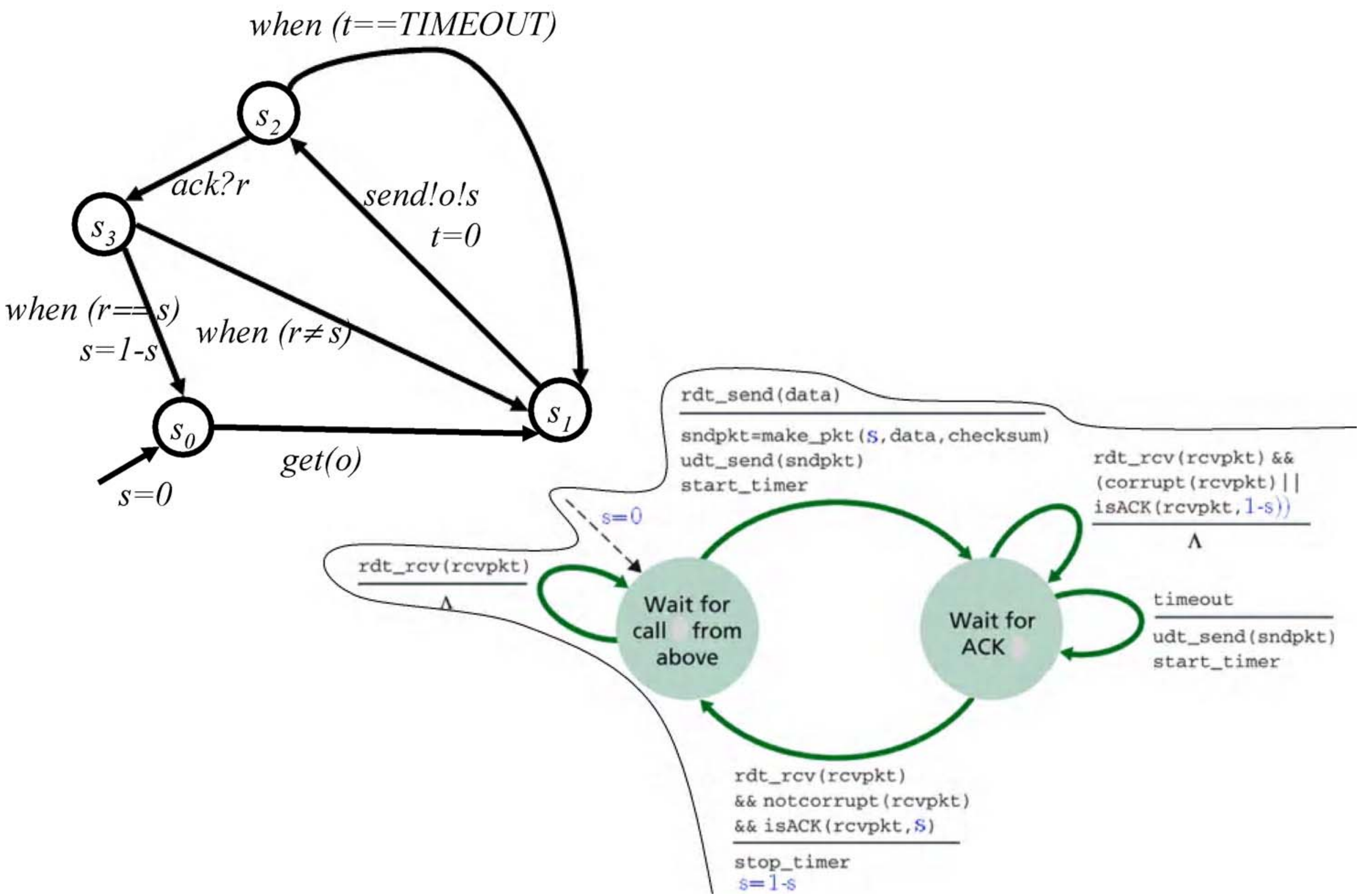
Step I: Folding the alternation bit



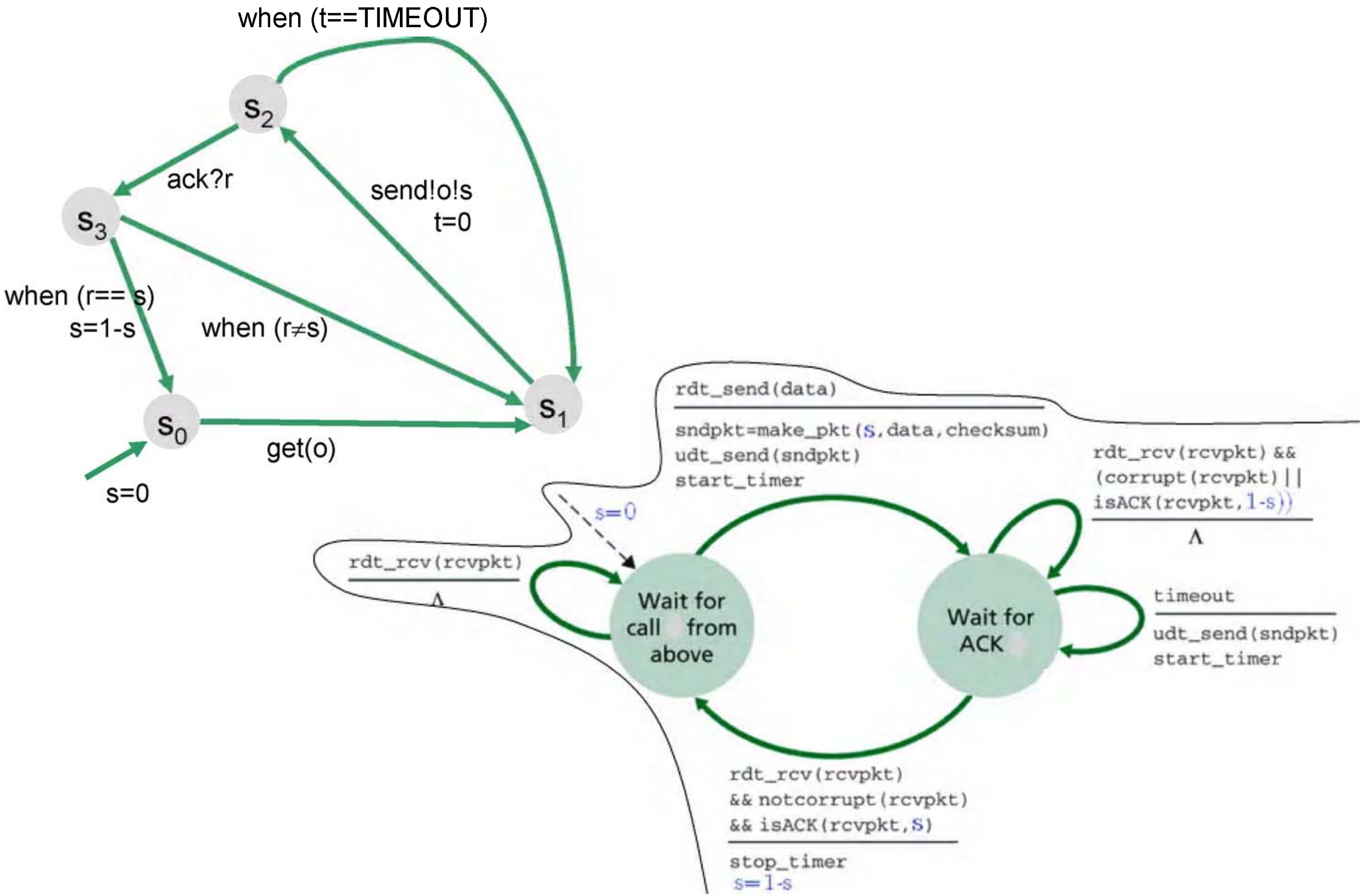
Step I: Folding the alternation bit



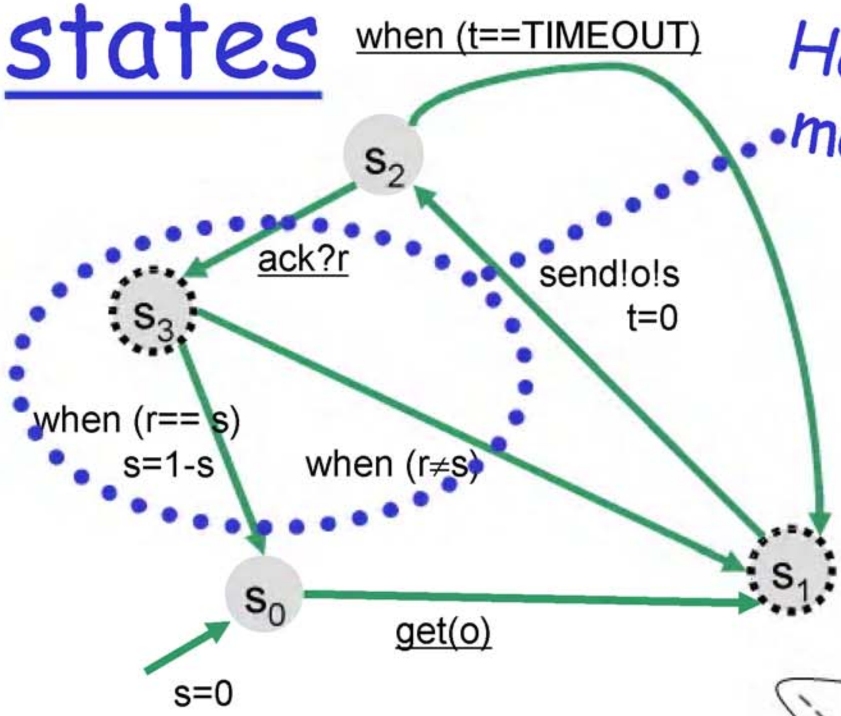
Step I: Folding the alternation bit



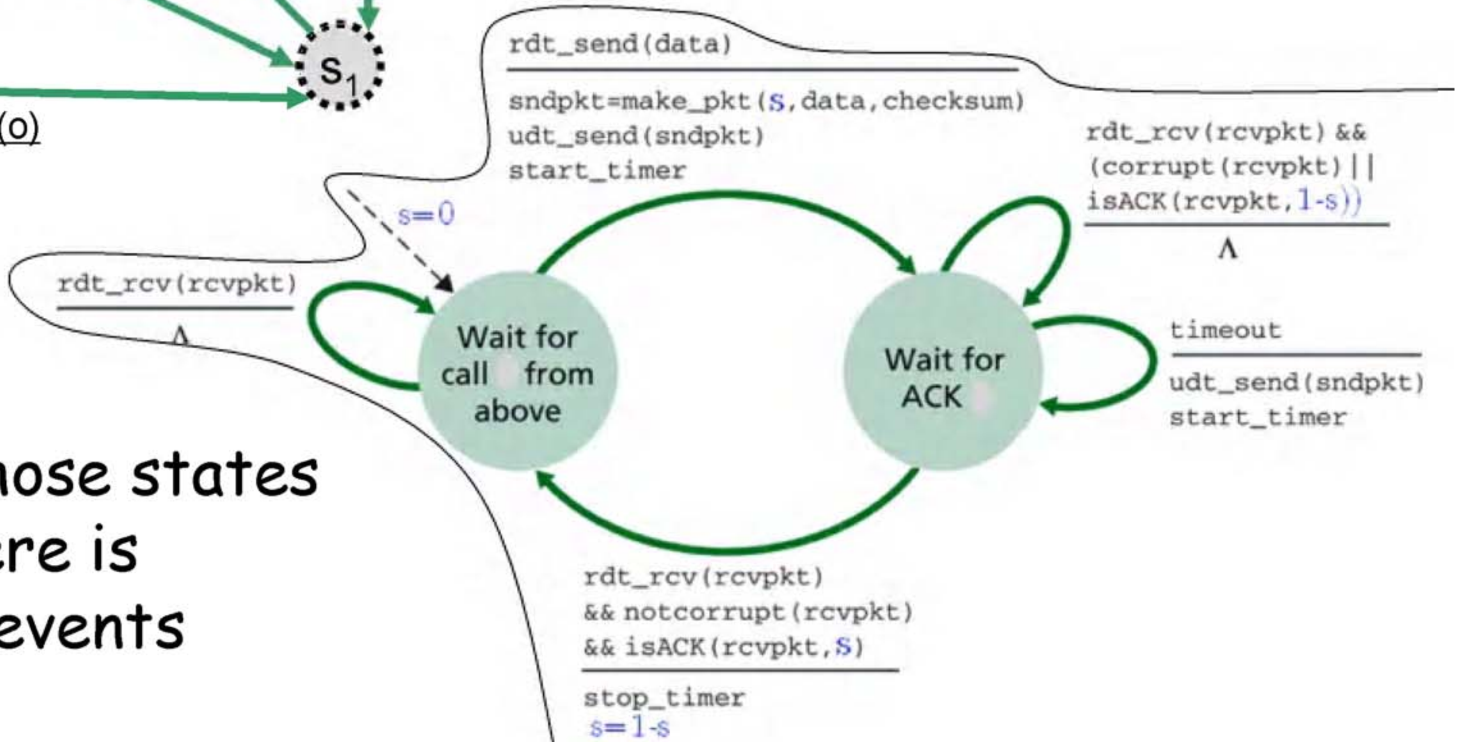
Step II: Fontification & Colouring



Step III: Getting rid of 'immaterial' states

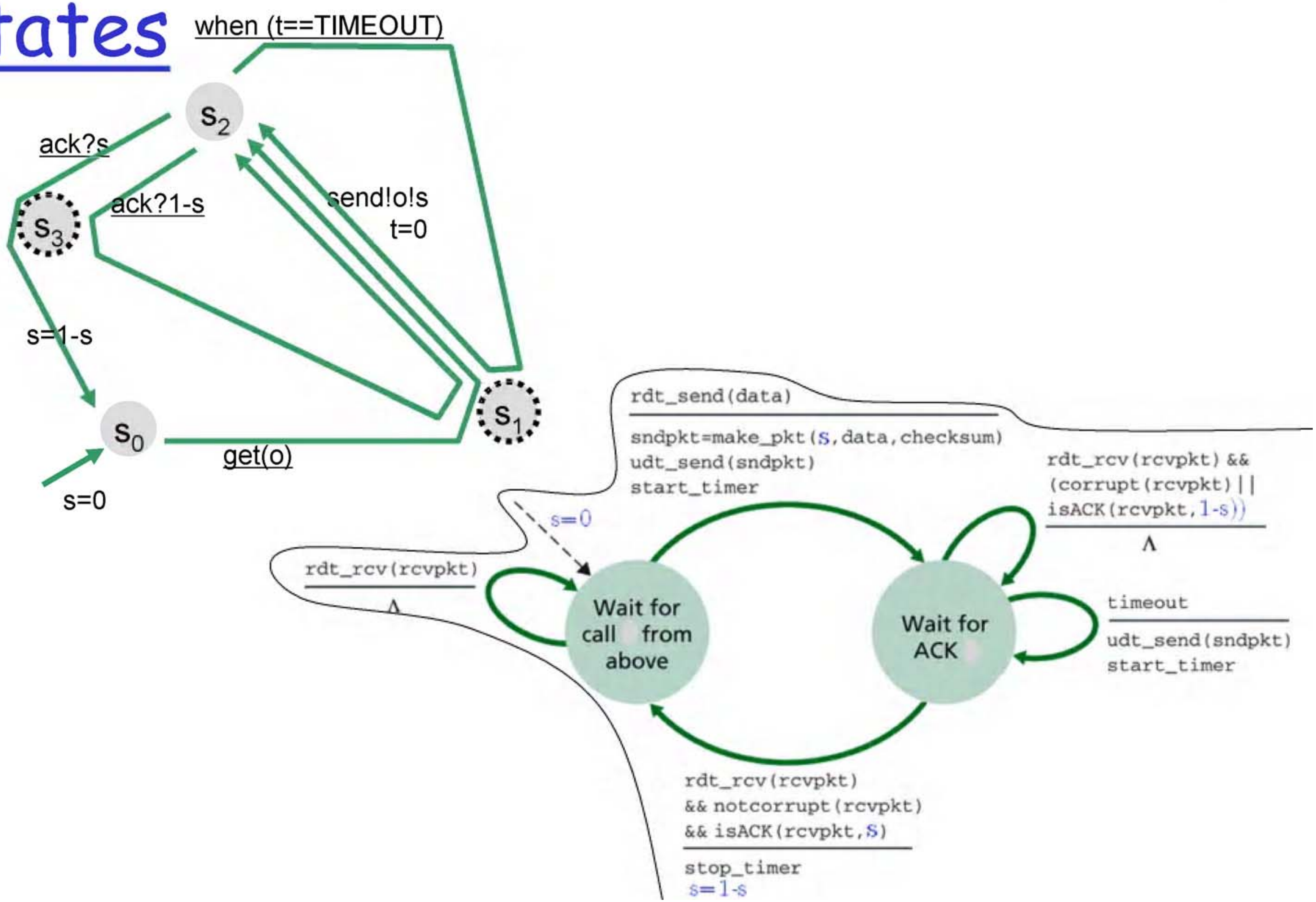


Here there is a bit of a catch which may need some separate discussion

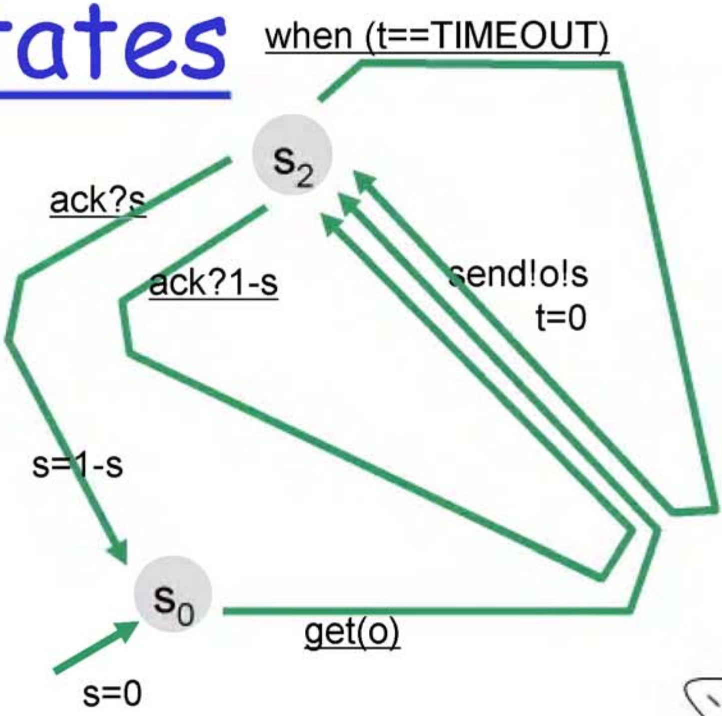


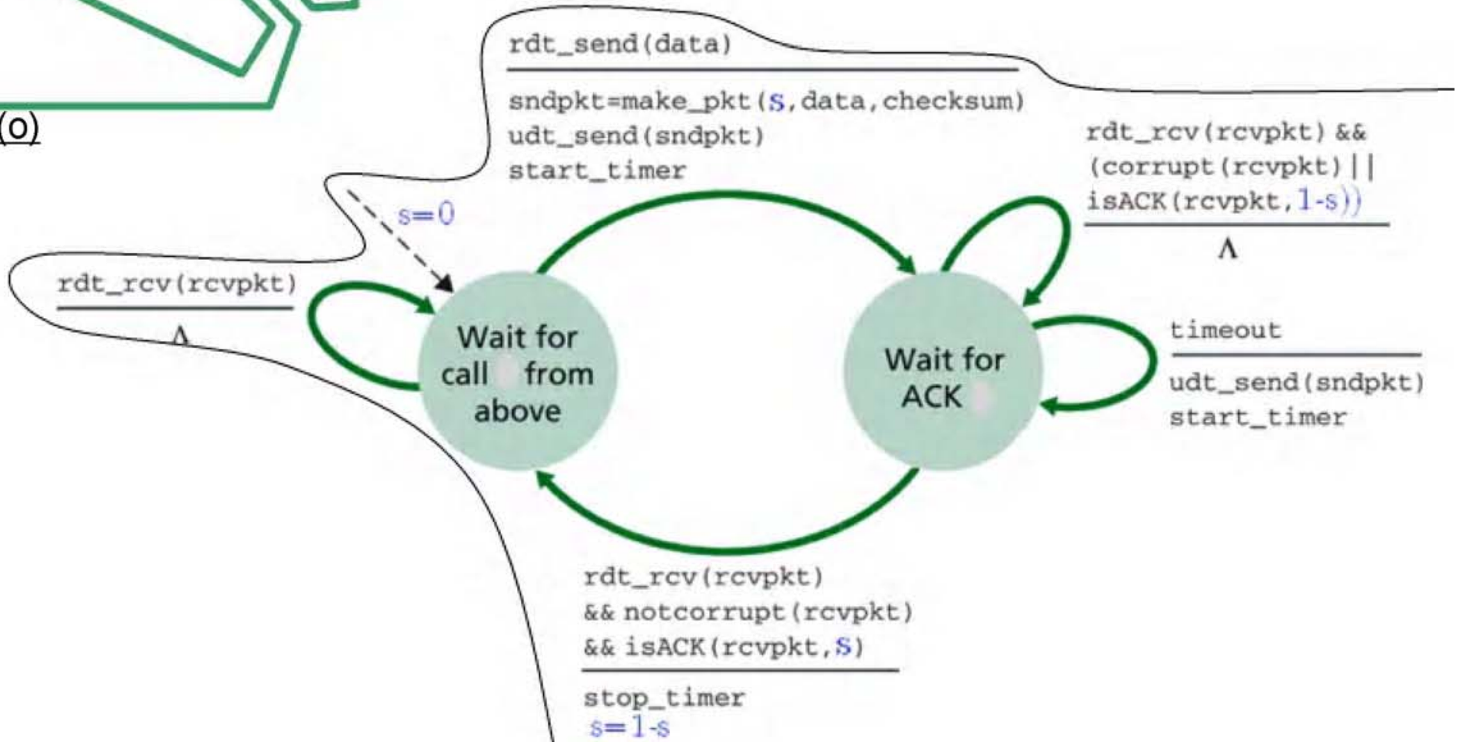
Keep only those states in which there is waiting for events

Step III: Getting rid of 'immaterial' states



Step III: Getting rid of 'immaterial' states





A note on value passing and value testing

- The principal use of

<name> ? <v_1> ? <v_2> ? ... ? <v_n>

variables!

is to receive data values via

<name> ! <d_1> ! <d_2> ! ..... ? <d_n>

values!

- You may also write this as

<name> ? (<v_1>,<v_2>,...,<v_n>)

and

<name> ! (<d_1>,<d_2>,...,<d_n>)

- However you write it, after a successful reception, the variables $v_1, \ldots, v_n$ hold the values $d_1, \ldots, d_n$

This is called value passing.

- One may also mix variables and constants, as in

<name> ? <v_1> ? <d> ? <v_2> ?...? <v_n>

- In this case communication is only sucessful if the very same 'd' is offered by the sender, as in

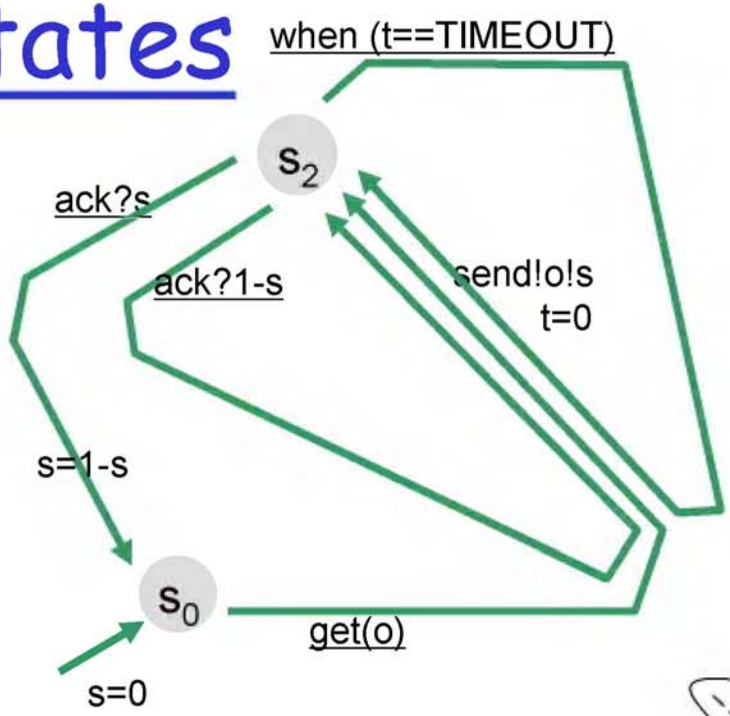
<name> ! <d_1> ! <d> ! <d_2> ! ... ! <d_n>

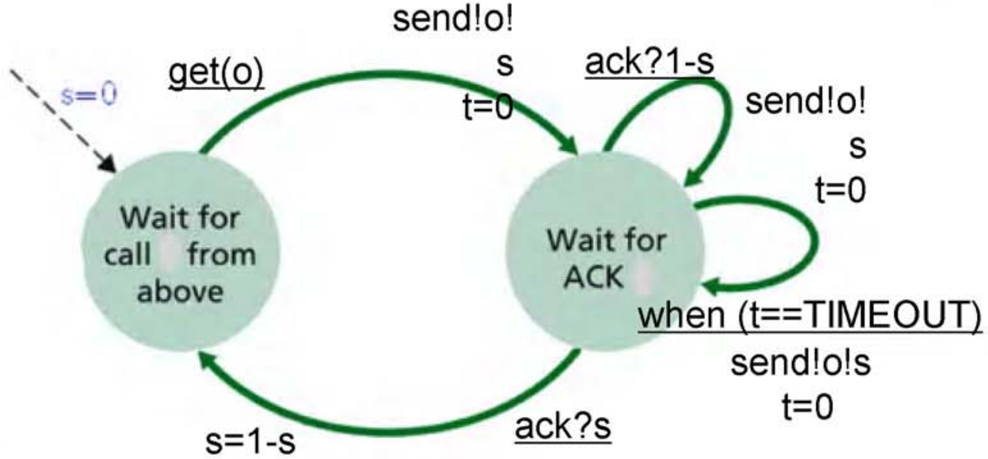
- If the offered and the expected 'd's are different, communication is impossible, and no state transition will be taken.

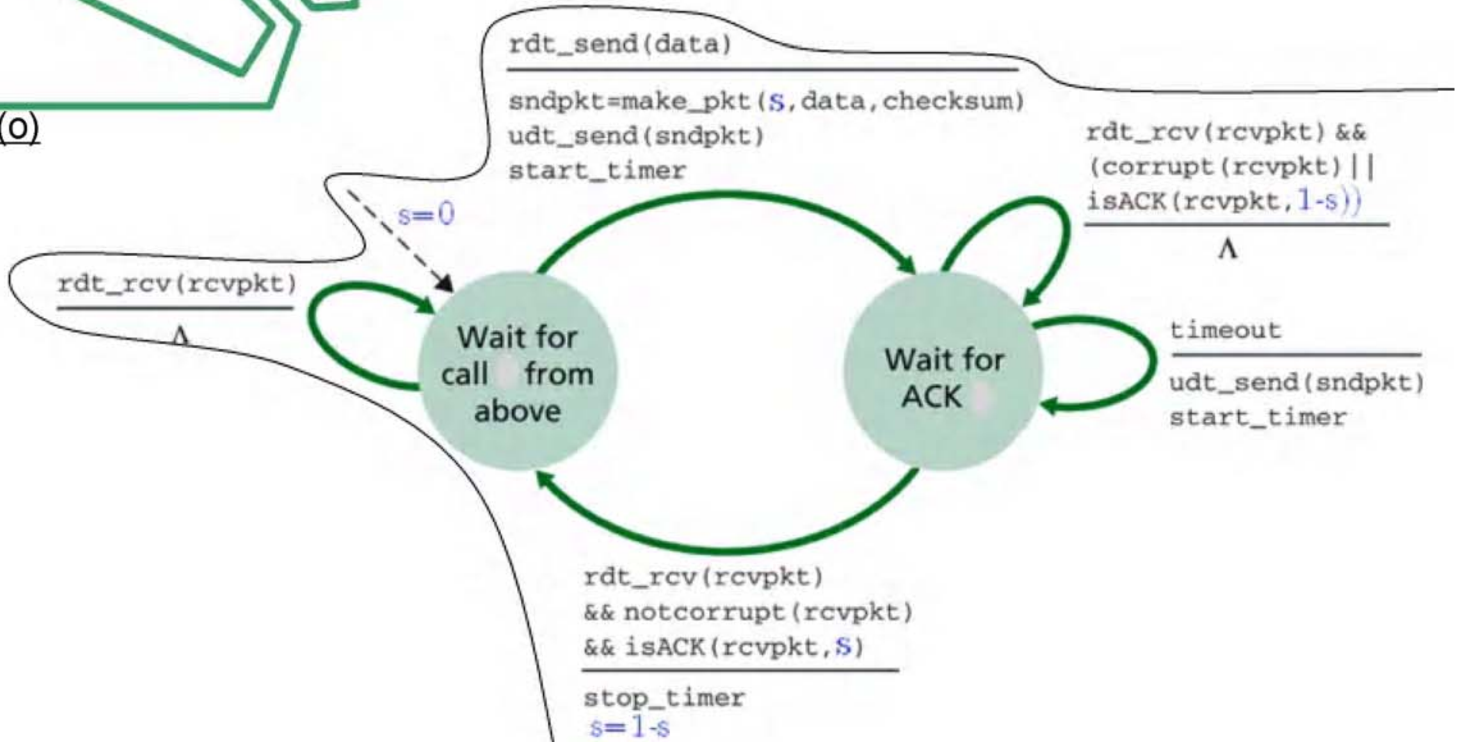
This is called value testing.

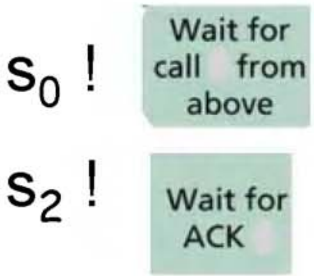
More on this e.g. in [Holzmann 91/03]

Step IV: Relabelling and rearranging states









Step V: Relabelling transitions

get(o) ! `rdt_send(data)`

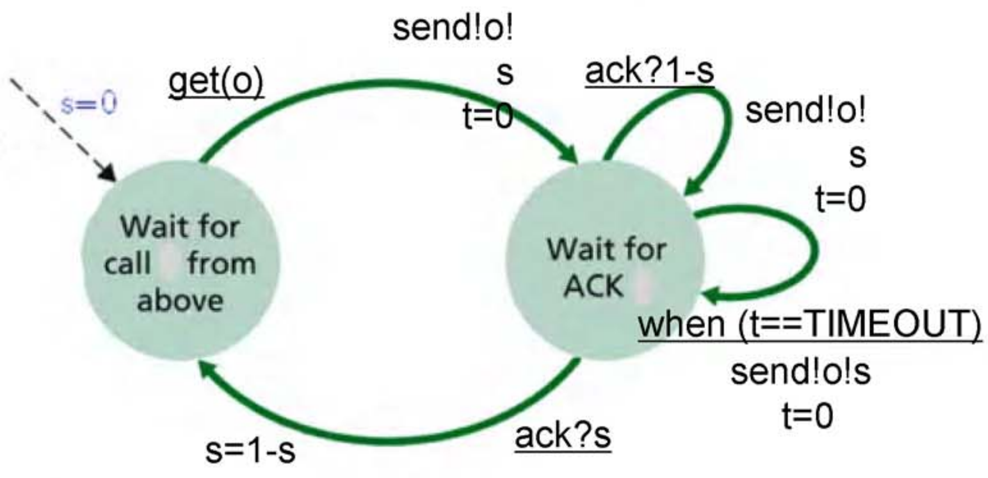
send ! o ! s ! `sndpkt=make_pkt(S,data,checksum)` `udt_send(sndpkt)`

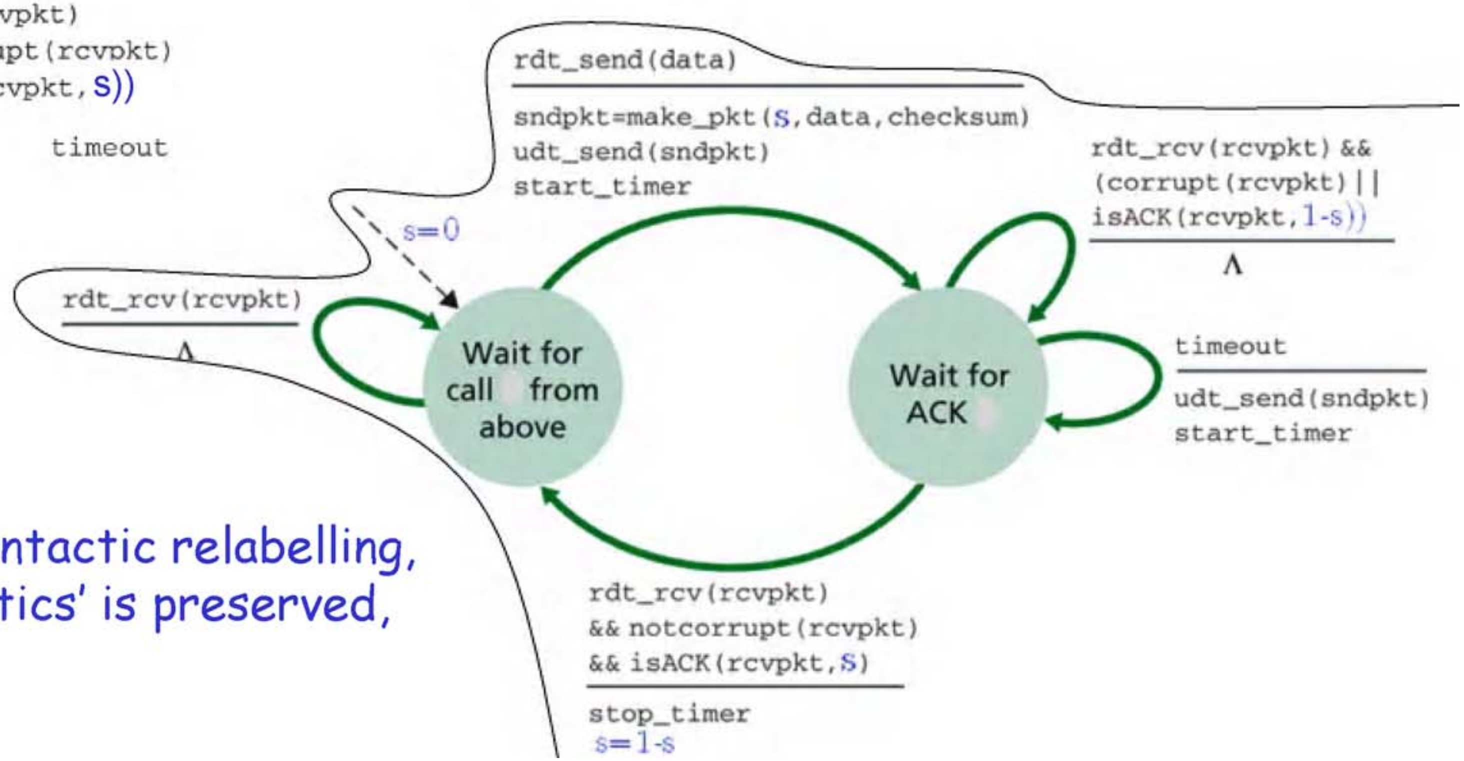
t = 0 ! `start_timer`

ack ? 1-s ! `rdt_rcv(rcvpkt) && (corrupt(rcvpkt)|| isACK(rcvpkt,1-s))`

ack ? s ! `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,S))`

when (t==TIMEOUT) ! `timeout`

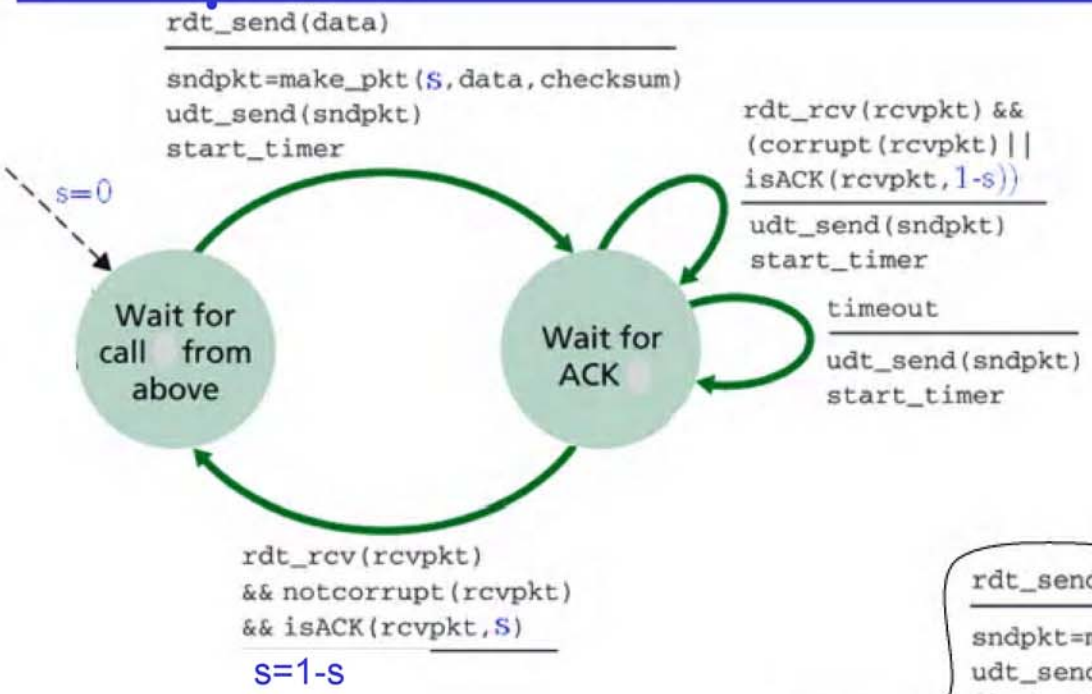




Note:
this is more than syntactic relabelling,
the 'intuitive semantics' is preserved,
for instance:

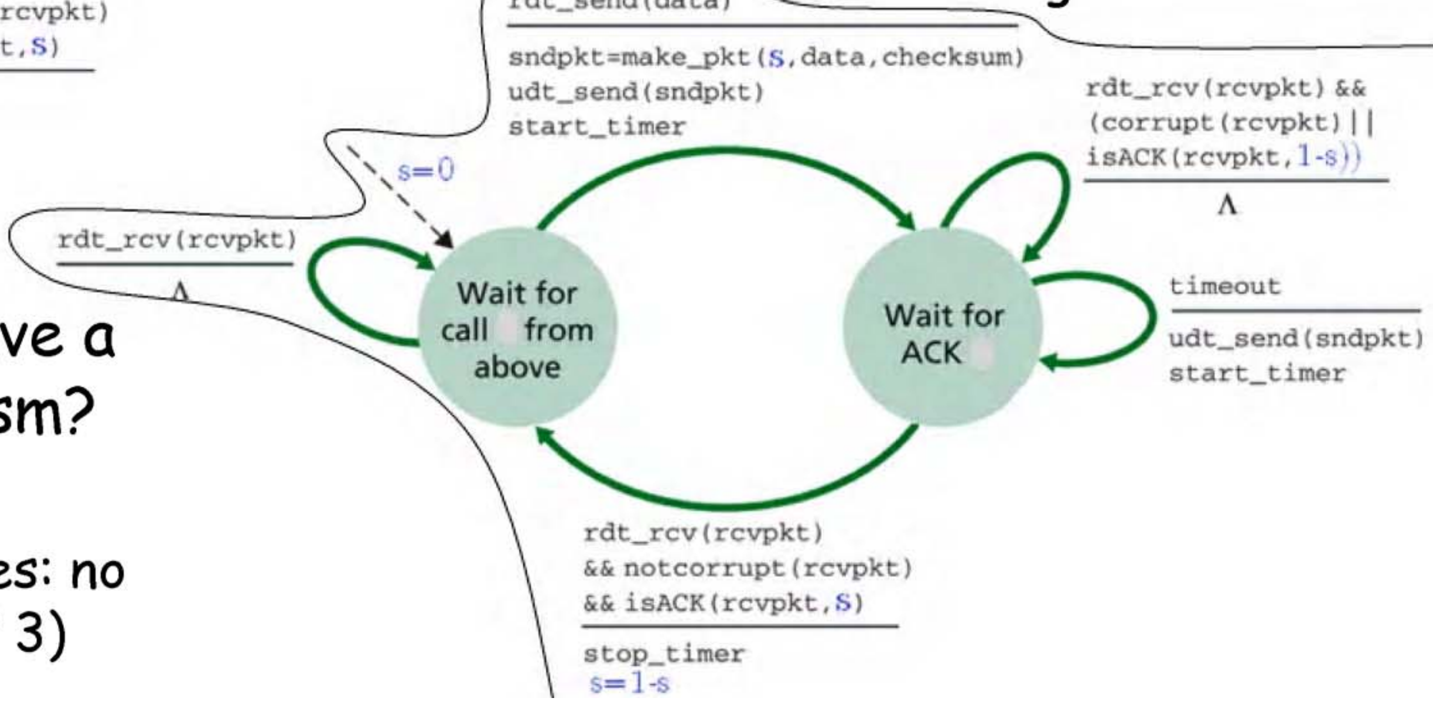
o ! `data`

Step VI: Check result



Find three differences?

1. no account for unexpected pkts when nothing to send
2. no timer stopping
3. no wait for timeout if wrong ACK received



- So, do we have a homomorphism?
  - Traces: yes
  - Timed traces: no (because of 3)

rdt3.0 sender of [Kurose Ross]: The A-B protocol